Operating Systems And Resource Reservations

Antonio Mancina

ReTiS Lab

Scuola Superiore S.Anna, Pisa

Ph.D. Thesis

03 April 2009

Tutor: Giuseppe Lipari .....

Supervisor: Tommaso Cucinotta .....

I would like to dedicate this thesis to several people.

First of all, to my girlfriend, Monica, who (almost) always supports my decisions and helps me a lot, even with my work, if and when possible.

Secondly, to my family which patiently put up with me in every situation (but I know it's an easy task, I'm a wonderful son and a really nice brother!).

Finally, to every single person belonging to the ReTiS Lab (and, recently, to the CEIIC building in general) thanks to which I've spent the most pleasant years of my life.

Thank you all, folks!

Acknowledgements

I would like to acknowledge several people, hoping not to forget anyone. First of all, my tutor, Giuseppe Lipari who, although being pressed by several obligations from his work, always found sufficient time to dedicate to me. Secondly, the ReTiS OS group, made by Luca Marzario, Michael Trimarchi, Tommaso Cucinotta, Fabio Checconi and, recently, Dario Faggioli that helped me a lot in every situation (not necessarily inherent to my research); due to his very recent efforts to be accepted (and even appreciated) by this group, I'll list here Marko Bertogna, who recently got married: my best wishes, Marko and Lidia! Finally, the group of friends and special people I met during the years I spent here: Luigi Palopoli, Giorgio Buttazzo, Marco Di Natale, Paolo Ancilotti, Mara Gandolfo, Enrico Bini, Paolo Gai, Giacomo Guidi, Davide Pagnin, Paolo Valente, Michele Cirinei, Igor Barsanti, Gabriele Cecchetti, Annalina Ruscelli, Luca Santinelli, Mangesh Chitnis, Yifan Wu, Paolo Pagano, Claudio Scordino, Nicola Serreli.

I met many other people during the last five years, and I can say, as a bullet-proof concept, that the Real-Time System Lab is one of the best work-places which I have ever been part of.

Abstract

Nowadays, an ever increasing interest from the industrial world in real-time scheduling capable operating systems is driving many development efforts in the context of general purpose operating systems towards the introduction of specific mechanisms able to provide the system user with some real-time capabilities. The recent literature describes many different approaches and so lutions to this problam and this leads to a wide variety of mechanisms, each with its drawbacks and its advantages.

In this work the existing state of the art inherent to the application of real-time experiences and principles within modern operating systems is going to be analyzed.

Contributions of this thesis may be listed as follows:

- an implementation of an Earliest Deadline First scheduler along with an algorithm for resource sharing in the context of a modern monolithic kernel based operating system;
- an implementation of Resource Reservation algorithms within a modern microkernel based operating system;
- the design and a possible implementation of a taxonomy to describe the class of resource reservation algorithms and to help the researcher to conceive and implement new scheduling algorithms belonging to this class.

In the author's opinion the last point is a fundamental contribute to start enriching the current state of the art with a better and more unified way to face real-time issues in the context of general purpose operating systems.

Contents

List of Figures

Chapter 1

Introduction

General purpose operating systems (GPOS) are the piece of software that most commonly - directly or indirectly - people have to deal with. Many examples about OSes of very different nature can be brought to the reader attention and, very likely, many of them have been at least heard about, the result of this being a very practical knowledge of the environments that we are going to describe and analyze in deep details.

Nowadays, many actors from the industrial world are investing many development efforts towards exploiting these more general and accessible resources in order to be able to deploy such technologies within their plants.

In the very recent past, many embedded devices used to run very small operating systems or tailored applications (in the form of firmwares) suited to the specific needs or fields of application. This model has been applied for many years, since it guaranteed a solid development cycle and performances, although applicable in a very limited and specific domain.

In contrast to this flow, a more versatile approach is required in order to speed-up the development cycle. The most suitable way, in this sense, would be to exploit general purpose operating systems, stripped of unnecessary components in order to fit the typical embedded devices memory sizes and somewhat enhanced in order to provide timely guarantees. By doing so, it would be possible to get the following advantages:

- **production cost**:since OSes provide the platforms which they run on with

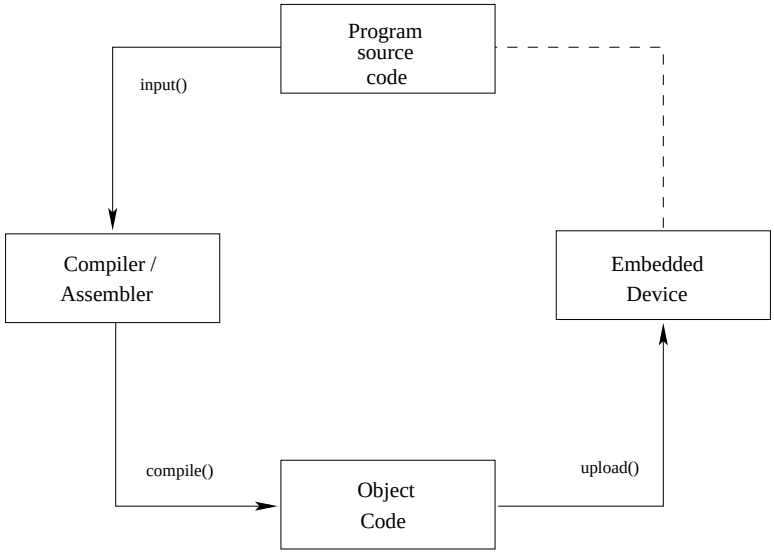

Figure 1.1: Typical embedded-platforms development cycle

all the facilities necessary to access every low-level built-on device, less time is spent on programming these low-level access routines;

- **time-to-market**: also the production time benefits from a general OS ported on the platform, because the developer may focus on the development of necessary application components only;
- **fault tolerance**: a GPOS helps the developer to make use of predefined services from a dependability point of view. Existing and widely adopted technologies allow one to spend less time in bug tracking and fixing.

In the industrial world, several Commercial Of The Shelves (COTS) solutions have been adopted by industrial companies. As such, there are expensive costs for acquisition, initial settlement and maintenance: these costs might be drastically reduced by adopting open-source, freeware solutions.

In this thesis we will focus especially on Operating Systems of the latter nature. Different commercial licences exist for this kind of products: since we are interested into the possibility of modifying the source code in order to tailor it according to our needs, this number gets definitely lower.

An important concept in the context of soft real-time systems is that of *quality of service* (QoS), whose definition is in tight relationship with the user satisfation at system and application level. Nowadays, many kinds of applications would directly benefit from an improved level of service, even though definitely not hard-guaranteed. Typical examples of such applications include multimedia video streamers, video games and VoIP communications.

In fact, for this kind of applications, it is often possible to single out a periodic nature of some processing threads. Hence, once the schedulability requirements of these threads have been analyzed, it is possible to let a real-time scheduler handle the taskset in order to provide temporal guarantees. Sometimes, in contrast, this is not sufficient: there are applications that, due to their intrinsically dynamic nature, may show a behaviour of wide-ranging nature. In this case, every attempt to build-up a system with these tasks may lead to dramatic decreases in the QoS as for temporary overruns or more generic misbehaviours.

In the context of this kind of applications, a more specific and robust class of real-time scheduling algorithms, namely *Resource Reservations*, was conceived in order to keep on providing temporal guarantees and, at the same time, isolating any task temporal behaviour, thus preventing misbehaving tasks from influencing all the other tasks' execution.

This thesis is organized as follows:

- Chapter 2 provides the reader with a detailed survey on the existing operating systems and how these OSes relate to the real-time scheduling domain;
- Chapter 3 describes the class of Resource Reservation scheduling algorithms along with the properties that each of these algorithms provides the scheduling system with;
- Chapter 4 gives some hints for implementing real-time scheduling algorithms within real modern operating systems of both monolithic and microkernel nature. At the same time we will describe a couple of practical experiences.

- Chapter 5 proposes a new taxonomy useful to describe the whole class of resource reservation algorithms in the most generic way as possible, thus easing the task to conceive, design and deploy new scheduling models.

In the last chapter some considerations on the topic of resource reservations and operating systems will be brought to the reader attention. I hope the reader will find this contribution of some, even small, importance in this huge domain.

Chapter 2

Operating Systems and Real-Time

This chapter will cover some important aspects of modern operating systems, considering the main alternatives a developer has to choose among, each with its positive and negative aspects. I will try to show how the architectural design at kernel and user level may affect the performances each OS offers with respect to the others.

Furthermore, a brief analysis of every solution will be conducted in terms of:

- inter-process communication (IPC);
- interrupt latency;
- scheduling-related latencies (context switches);
- dependability;
- real-time compliance.

At the end of this chapter, the reader should feel comfortable enough with the basic notions needed to understand in detail the remainder of this thesis.

Let me now define the basic terminology needed to describe the whole set of architectures which is going to be presented.

2.1 Terminology and general OS concepts

In this section, general and standard concepts in operating systems theory will be introduced.

2.1.1 Process, threads and task scheduling

The concept of *process* (or task - these terms are used interchangeably throughout the rest of this thesis) is fundamental in OS theory since it defines the basic schedulable entity within a multiprogramming environment.

It is often referred to as "an instance of a program in execution", whereas a program may be defined as an executable file (different environments are characterized by different formats and, possibly, extensions). When a program is written in a re-entrant way (i.e. it does not modify itself), it is possible to run multiple independent instances of it at the same time [61].

In many systems a *thread* is something different from a process, and it is someway contained into it. Several threads may exist in the context of a process and all of them share with it many environmental parameters. Several implementations exist for this concept and each unit is scheduled according to some thread manager's policy [35].

The concept of scheduling comes to describe the sequence of the activities which the kernel puts in action to select an entity, be it a thread or a process and start running it. It is very common to describe the states which a process may find itself into, during its existence, by using a state diagram like the one in Figure 2.1.

By using this diagram we can describe the whole life cycle of a process:

- a new process is created by means of a fork() system call (this holds for POSIX systems, every other OS has its own equivalent);
- after an initialization phase, it goes into the ready state;
- from this moment on, the process will undergo the scheduler decisions, and will behave correspondingly by switching its state back and forth among the possible ones shown in the diagram;

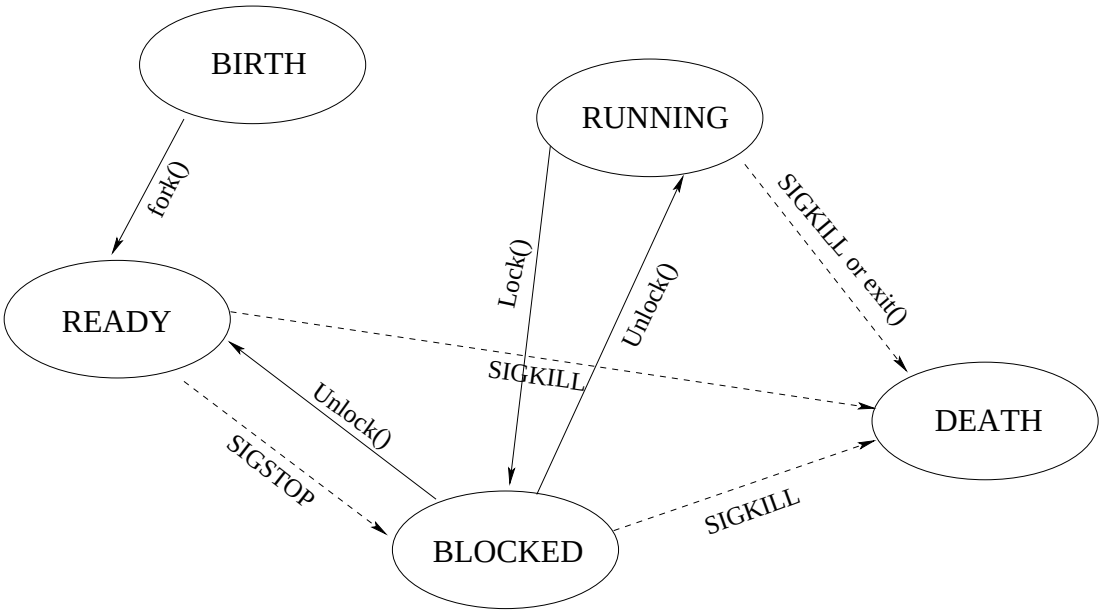

Figure 2.1: Process states diagram

- finally, either with a voluntary program exit statement (through an exit () or **return** instructions) or a possible fault condition, the process will reach the dying state and the system will take care of cleaning it up.

One of the most important features of a scheduling algorithm is the *preemptability*. If an algorithm is *preemptive*, then whenever a task whose priority is higher than the currently running one's becomes ready, the scheduler immediately stops the current task execution and starts the new one. This feature brings forth many problems when it comes to handle resources shared among different processes. We will briefly describe a general way to solve these issues in the following sections.

Many more versions of this diagram exist, each one trying to describe finer-grained processes' states transitions. For the purpose of this chapter we will limit our discussion to the described states only.

2.1.2 Inter-Process Communication

At a certain point during its life, a process may need to interact with other processes, running alongside it. In this section we will have a brief look to the most common implemented communication mechanisms within general purpose operating systems.

Three types of interaction among processes may be identified:

competition , used to describe the case when two or more processes need to access the very same resource in order to keep on executing;

cooperation , comprising all the situations in which two or more processes need to run in a coordinated way, possibly exchanging data;

synchronization , when to carry out its own work a process needs to wait for some conditions to hold true as from other tasks' execution.

As far as cooperation is concerned, in standard Unix-like monolithic kernels, the communication may take place by means of:

- pipes;
- shared memory;
- message queues;
- ...

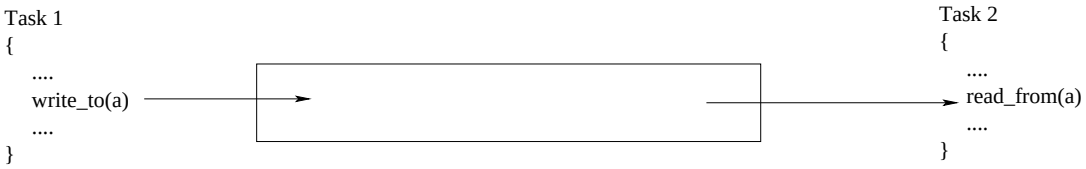

Figure 2.2: Communication through pipes

Pipes are used to implement a two-actors communication. Each process opens one of two ends, either in read or write mode, and starts filling it with data or picking the data out of it (see Figure 2.2).

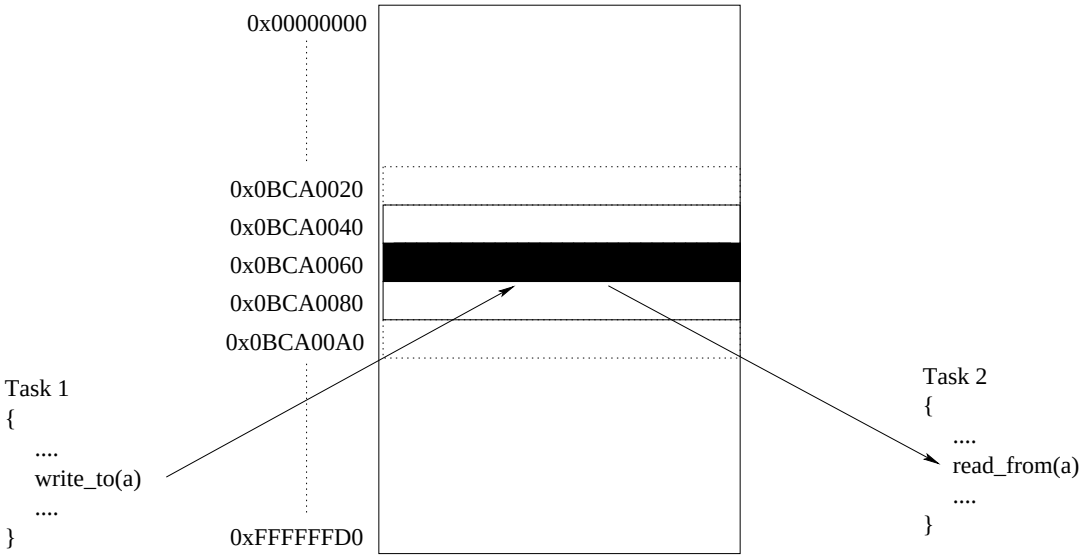

Figure 2.3: Communication through shared memory

Shared memory is the only communication facility a group of threads (part of the same process) may exploit to bring the communication off. It is carried out by using the very same memory areas, shared by one task's created threads (see Figure 2.3). It is also used in the context of standard process communication, through a system call which maps a memory area on a standard device file by means of which these processes may communicate.

Message queues are a mechanism resembling pretty much a distributed messaging facility, but with an asynchrounous nature. A process willing to send data to another one has to put a message within this queue. When the receiving process is ready to receive the data, it will issue a read operation on this queue, getting the message.

As far as synchronization and competition are concerned, **semaphores** come to solve this kind of problems. A semaphore is implemented as a primitive data type (usually an integer) exporting the following interface:

- every wait on it decreases its value by one;
- every signal on it increases its value by one;

Whenever its value reaches 0 a wait operation becomes a blocking primitive: the issuing process will be unblocked by another task's signal operation [34].

In systems of different nature, processes interaction may happen thanks to different mechanisms.

As an example, in microkernel based OSes this communication is typically based on message passing, which is a synchronous facility. As such, many classic synchronization problems are automatically solved by the explicit rendez-vous the involved processes have to run against, for the communication to take place.

2.1.3 Resources Sharing

A modern system is comprised of many peripherals and devices whose role is to provide the end user with a vast amount of important services. A brief and definitely not exhaustive list of them might include:

- CPU;
- volatile memory;
- permanent storage;
- network device(s);
- screens and printers;
- ...

In a multiprogrammed environment, several processes might need to access each of these devices at the very same time, giving birth to many hardware and software conflicts. Due to the nature of common devices, an arbitrated access protocol is needed in order to provide the needing tasks with a dedicated service.

Semaphores, *mutexes* and *monitors* are among the most common mechanisms an operating system puts in place to control this shared access. These concepts are of paramount importance for interactions to be correctly carried out.

Mutexes are a special kind of semaphores, with initial value set to 1 (and for this reason, called *binary semaphores* as well). They are used to protect the access to a shared resource in the context of which only one process at a time may operate.

Monitors are a higher level synchronization and protection facility which automatically takes care of locking and unlocking operations while a process accesses its protected resource.

We will see how to deal with these synchronization issues when the system has to provide real-time guarantees.

2.1.4 Interrupts and system calls

Since a system is "a set of interacting or interdependent entities, real or abstract, forming an integrated whole" [24], we need a way for this whole to interact with the external environment. These interactions may be correctly carried out thanks to the mechanisms of **interrupts**.

In Figure 2.4 we see how the interrupt system is usually implemented at hardware level. Each device has a control and data bus in common (i.e. the system bus). When a process asks for service from a device, it does it asynchronously. Then the device starts working on behalf of it and, whenever it is ready to give the result back to the requiring process, it signals the interrupt control on its dedicated IRQ line. At this point the interrupt controller notifies the CPU of data being ready and of the source IRQ number. The CPU will address directly the source of interrupt to transfer the data towards memory or to immediately utilize it [61].

Whenever a process requires a service from the underlying operating system, it issues a *system call*. A syscall is a special istruction issued by a task which makes the execution switch from user to kernel context (we are not going into details about how this switch occurs, since it is architecture-dependent and out of the scope of this dissertation). Thus, the CPU starts executing code at the highest privilege level on behalf of the calling process, in order to serve its needs. Eventually, the original process context will be restored and the process will start its execution over.

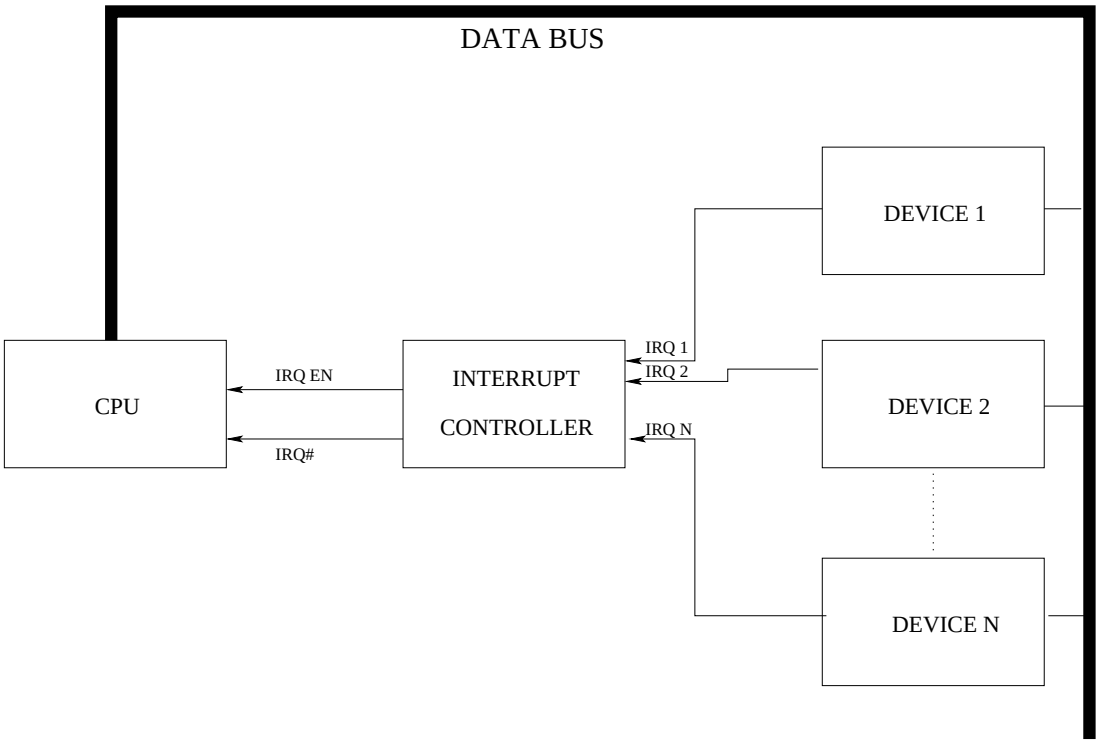

Figure 2.4: Interrupts system implementation

2.2 Standard programming and communication interfaces

When a kernel programmer wants to implement new features within the system kernel, he has to make use of a specific Application Programming Interface (API), exported by the operating system.

Although the original kernel designer has the complete freedom to conceive and design the operating system as he prefers, if he decides to make its interface attain to a recognized and official programming standard, he will favour the application portability between his and other operating systems.

We will look at some of these standards in the following sections. At the end of this chapter the reader will be presented a brief survey of existing operating systems and an analysis of their conformance to these standards.

2.2.1 The Portable Operating System Interface

POSIX [23] is a collective name of a family of standards conceived by the IEEE to define the Application Programming Interface for all the software compatible with the Unix operating systems family. In particular, the user and software interfaces to an operating systems are defined in 17 different documents.

Currently POSIX documentation is divided in three parts:

- kernel APIs (which include real-time services, threads interfaces, real-time extensions, security interfaces and network access and communication);
- commands and utilities;
- conformance testing.

2.2.2 Osek/VDX

Osek/VDX [14] is a group of standards that produced specifications for an embedded operating system, a communication stack and a network management protocol for embedded devices in the automotive field. The two initially separated projects have been conceived by two groups of german and french automotive

industrial groups and eventually merged into a unique specification which is the de-facto standard for all the electronics and communication within a car.

2.2.3 ARINC

The Aeronautical Radio, Incorporated (ARINC) [4] association is one of the leading providers of transport communications and system engineering solutions for eight major industries: aviation, airports, defense, government, healthcare, networks, security, and transportation.

It provided the engineering community with many different standards: as far as our analysis is concerned their most important contribution is contained in the ARINC-653 standard [5], which describes the partitioning of computer resources in the time and space domains. This standard also specifies APIs for abstraction of the application from the underlying hardware and software.

2.2.4 ITRON

Conceived in Japan in 1984, The Real-time Operating system Nucleus (TRON) is a set of interfaces and os design guidelines. Most japanese embedded devices currently adhere to this standard.

Several subprojects forked from the main line, with slightly different goals and market customers. Its first and most important derivative, Industrial-TRON (ITRON), is an open-source specification for real-time operating systems architectures targeted at embedded systems.

In 2003, Montavista got a partnership with the T-Engine forum with the goal to create a universal standard for embedded platforms real-time operating systems. To the best of the author's knowledge this effort is still undergoing.

2.3 Real-Time Scheduling

This section will briefly introduce some key aspects of real-time theory.

In first place, let me introduce the basic terminology and model that are going to be utilized throughout the following sections.

A *task* T_i (or *process*) is comprised of a sequence of *jobs* $J_{i,j}$, each of which is described by at least an *arrival time* $a_{i,j}$ (or *release time* $r_{i,j}$), an *execution time* $c_{i,j}$ and an *absolute deadline* $d_{i,j}$, that is the time by which the execution of the current job must be carried out (often we will speak of a *relative* deadline, that is $D_{i,j}$ time units after the corresponding release time $r_{i,j}$) (see Figure 2.5).

Figure 2.5: A graphical representation of the important task parameters.

This very generic model may be specialized according to the tasks' nature:

periodic: each job has $r_{i,j+1} = r_{i,j} + T_i$ where T_i is the task period and in case of *implicit* deadlines, $d_{i,j} = r_{i,j} + T_i$;

sporadic: each job has a variable execution time and arrival time so that in place of them we introduce the concepts of *Worst-case Execution Time* and *Minimum Inter-arrival Time* which all the guarantees must be built upon;

aperiodic: typical of one-shot job executions, when a task appears and soon after dies.

Some important concepts may be now defined:

Lemma 2.3.1. *A taskset is schedulable according to an algorithm* Γ *if and only if there exists a placement of every tasks' job such that every one completes its execution within its own deadline.*

Lemma 2.3.2. *A taskset is feasible if there exists at least an algorithm* Γ *according to which the taskset is schedulable*

In 1973 Liu and Layland published a paper [45] which has become the base for the real-time scheduling theory. Their key contributions were manyfold:

- a first standardization for modelling real-time tasks was proposed;

- both the standard uni-processor fixed and dynamic priorities scheduling algorithms were introduced and analyzed.

Rate Monotonic and **Earliest Deadline First** are still the most adopted scheduling policies in real-time systems due to their reliability and deep understanding and we will go through their definitions and some examples in the following section.

Task	C	D
A	2	6
B	1	4
C	2	8

Figure 2.6: Sample task set used in this section

2.3.1 Classic Theory

The algorithms considered in the following paragraphs belong to the class of full-preemptive ones. Non-preemptive variants do exist but they are out of the scope of this brief presentation and they will not be presented.

Rate Monotonic Let us start with the de-facto standard for fixed priority scheduling algorithms, Rate Monotonic. It assigns a priority to the tasks it schedules according to the following rule:

the shorter the period of a task, the higher its priority.

This very simple rule makes possible to implement the priority assignments as a static off-line procedure. This has the advantage of easing the scheduler activity when it comes to select the new task to run, since it is sufficient to maintain a statically allocated table in memory.

The RM algorithm has an inconvenience, though. Since there is no necessary and sufficient condition to test the schedulability of a taskset, but there is only a sufficient condition, the risk to reject a possibly schedulable taskset is not null. Liu and Layland determined the following sufficient condition to declare a taskset as schedulable:

Theorem 2.3.3. *A taskset is schedulable according to the Rate Monotonic algorithm if*

$$\sum_{i=1}^{n} U_i \leq n(2^{1/n} - 1) \tag{2.1}$$

which, in the worst case of n going to ∞ becomes:

$$\lim_{n \to \infty} n(2^{1/n} - 1) = \ln 2 \simeq 0.69 \tag{2.2}$$

Theorem 2.3.3 means that, for the admission test to accept a taskset in the system, the total CPU load must not get higher than the value of Equation 2.2.

Figure 2.7 represents the resulting schedule from the sample taskset of Table 2.6.

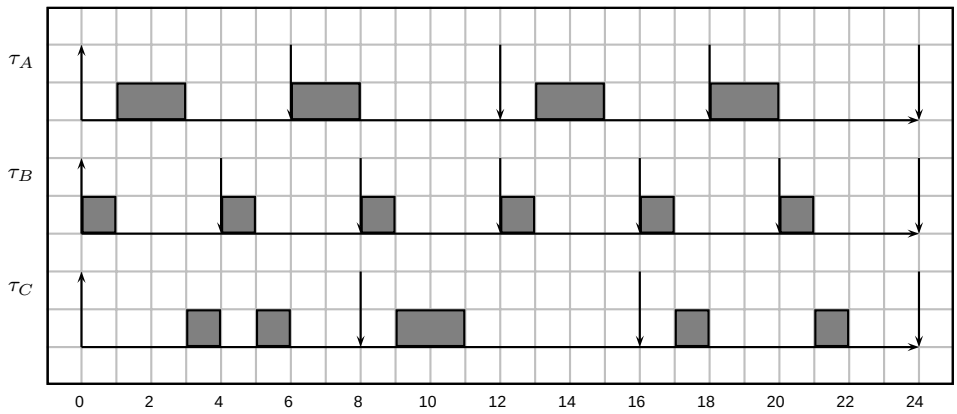

Figure 2.7: The scheduling sequence obtained from a Rate Monotonic scheduler

Earliest Deadline First To overcome the drawbacks of Rate Monotonic, Liu and Layland proposed the EDF algorithm. The scheduler selects the next process to schedule according to the following rule:

the earlier the current absolute deadline of a task, the higher its priority.

The EDF schedulability condition, in contrast with RM, is a necessary and sufficient one.

Theorem 2.3.4. *A taskset is schedulable according to the Rate Monotonic algorithm if*

$$\sum_{n=1}^{\infty} U_i \leq 1 \tag{2.3}$$

This means that, for a taskset to be schedulable, it may be worth all the CPU utilization. With EDF no CPU bandwidth gets wasted, so a higher number of taskset may be accepted by the admission test.

On the other hand, a higher complexity is required when the scheduler comes to select the new process to run, because it has to maintain a dynamically updated list of increasing absolute deadlines and pick always the first from it.

In Figure 2.8 we may observe how the resulting schedule of the same taskset as before, differs from the RM case.

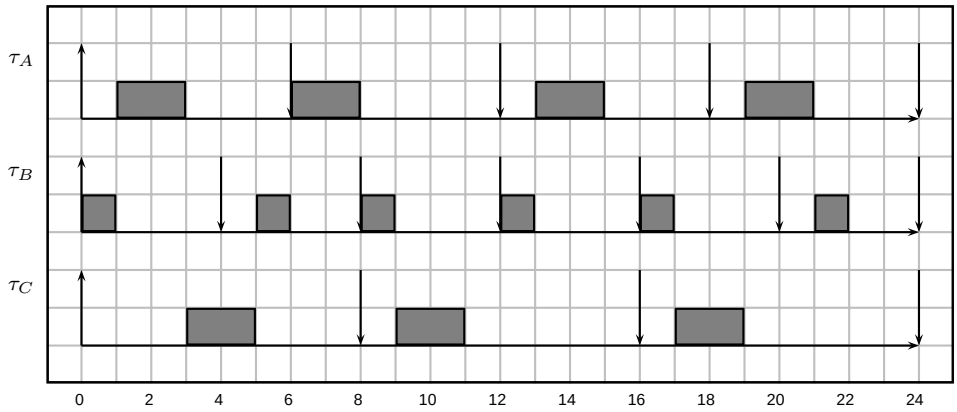

Figure 2.8: The scheduling sequence obtained from an Earliest Deadline First scheduler

For a deeper comparison between RM and EDF we refer the interested reader to more specific publications ([32]).

2.3.2 Quality of Service

An important concept, somewhat orthogonal to the classic real-time theory, is tightly related to the user perception of the system behaviour, and it is usually referred to as *Quality of Service*.

QoS is an important property of a system, especially when it comes to deal with multimedia applications, like movies or videogames. This kind of applications may be placed in the soft real-time domain, getting benefit from temporal guarantees but not jeopardizing the whole system integrity in case of deadline misses.

A lot of theory has been developed behind this concept, starting from syntaxes to describe quality requirements, up to several metrics to represent the service quality provided. For the purpose of this discussion, we will limit to a qualitative approach, coping with QoS metrics from a real-time stanpoint only.

In particular, an important measure of this parameter is tightly bound to the concept of *lateness*, defined as follows:

$$L_{i,j} = f_{i,j} - d_{i,j} \tag{2.4}$$

where $f_{i,j}$ is the finishing time of the j-th job of the i-th task. When this parameter is positive, the job completed its computation with a certain delay with respect to its current deadline and this concept, within a soft real-time system, leads directly to a degradation of the perceived quality from the user perspective.

Once the lateness has been defined, we may define the *tardiness* of the j-th job of the i-th task:

$$E_{i,j} = \max\{0, L_{i,j}\} \tag{2.5}$$

It is sometimes convenient to relate the tardiness of a job to the task period, thus obtaining the *normalized* tardiness, defined as follows:

$$E_{i,j} = \frac{\max\{0, L_{i,j}\}}{T_i} \tag{2.6}$$

By keeping track of these parameters, it is even possible to infer how well a system is behaving during its working, possibly deploying the necessary countermeasures to face any possible misbehaviours.

2.4 Brief analysis of existing Operating Systems

In this section, a selection of the most important and widely adopted Operating Systems, in authors' opinion, will be analyzed.

We will propose a partitioning of these systems according to three important macro areas:

- General Purpose Operating Systems;
- Operating systems for embedded devices;
- Real-Time operating systems.

2.4.1 General purpose operating systems

This area covers, by far, the broadest range of possible alternatives in terms of architectures supported and features offered to the end user. Nowadays, the choice has to be made among three possible big groups of candidates:

- MS Windows series;
- Unix-like operating systems (*BSD, Linux, Minix, ...);
- Mac OS X.

One way to further partition these candidates is according to the nature of the underlying kernel.

Monolithic typical of all modern Windows and Unix systems, it usually features two privilege levels only, kernel and user level. At the highest level, all the typical kernel components run on behalf of the user-level applications that made request of service. All system services and device drivers run in the very same context, thus offering the highest performances together with a high risk level as far as the overall system dependability is concerned (see Figure 2.9).

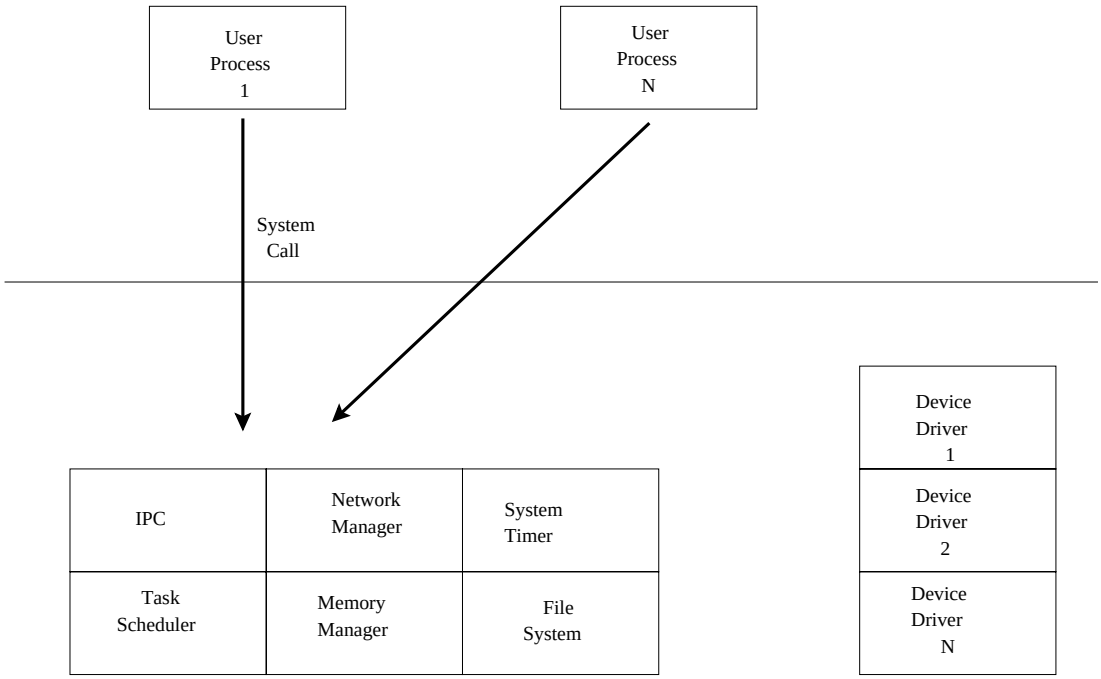

Figure 2.9: Monolithic kernel structure

Microkernel found in older OS and, under certain circumstances, in modern versions of old operating systems (e.g. Minix 3). All the ring levels (i.e. the privilege levels) are used and the kernel in the former sense does not exist any more. At the highest level, only the basic mechanisms can be found: IPC, process scheduling, system timing and kernel calls handling. Then, moving towards lower privilege levels we find the device drivers section, the servers section providing typical, higher level system services, as network and filesystem and, finally, the user space level (see Figure 2.10).

The main difference among these operating systems concerns their availability, being the second group free of charge and often, but not always, open source, thus allowing everyone to develope new features and tailor the system according to his own needs.

Another important point is the adherence to the standard *POSIX* (see Section 2.2.1). This standard defines the interface (or API) the developer has to use to ask for system services and to develop new features at kernel level. Unix-like

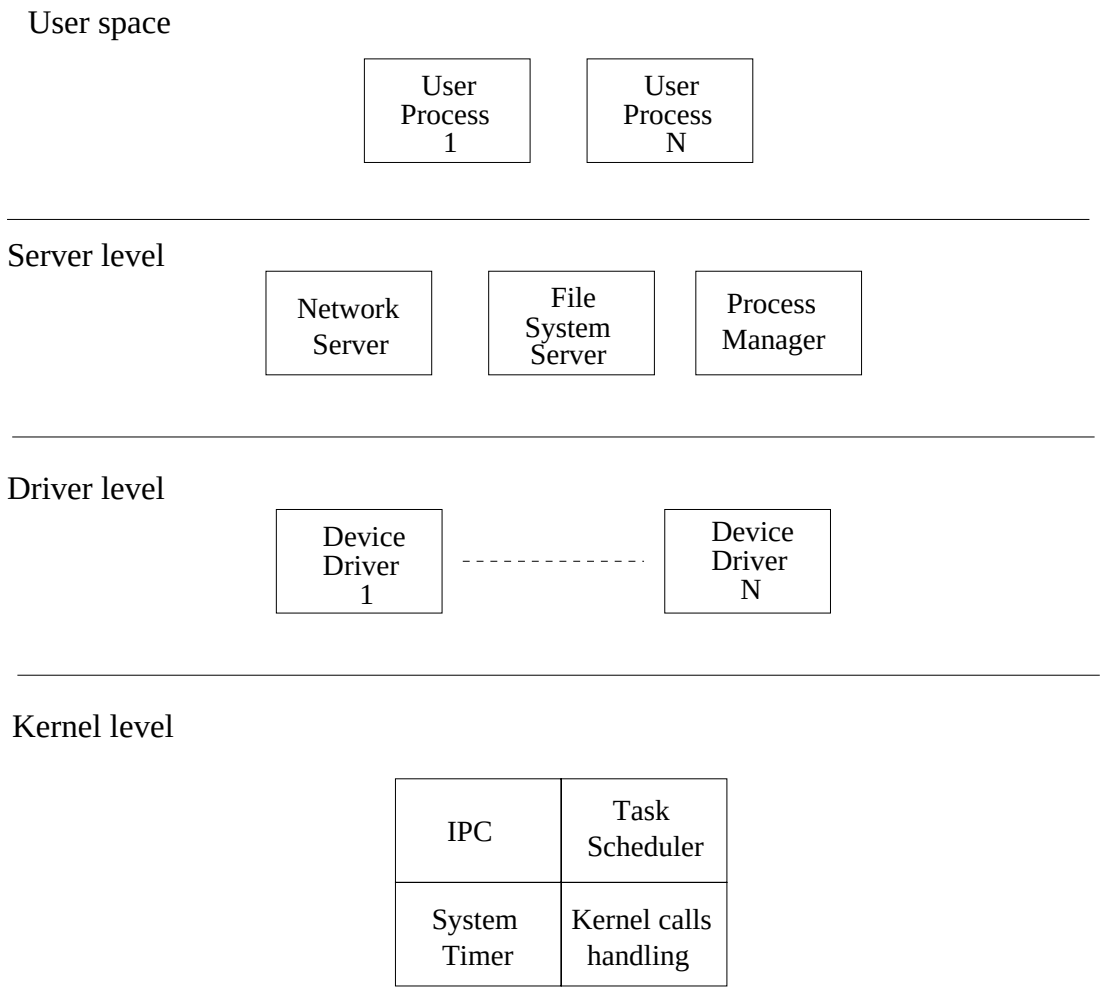

Figure 2.10: Minix 3 architecture (microkernel-like)

OSes are more likely fully or almost fully compliant to the standard, whereas Windows systems need a software layer interposed in between, like Cygwin [6].

Windows Vista [12] is the current release of the famous operating system from Microsoft. As far as its kernel is concerned, a hybrid nature (mixed monolithic and microkernel aspects) is the better way to describe it. Since Windows operating systems are closed source, though, it is not obvious as to what extent the microkernel nature operates (windows systems have tradionally been monolithic-shaped).

In the current release many new features have been introduced, most notably a new process model, which includes the *protected* mode. When a normal process wants to access a protected process space, it must have special privileges to do that (i.e. a debugger willing to take control over another process).

Mac OS X Leopard [3] is the last version of the succesful operating system from Apple. It features the *XNU* kernel, a special architecture originating from a combination of the Mach kernel plus some FreeBSD-4.3 components. Mach is a micro-kernel based operating system and, as such, it provides the system with all the *pros* and *cons* of such an architecture. In particular, its augmented dependability with respect to entirely monolithic kernels allows one to utilize Leopard within multimedia domains achieving a very stable production environment.

2.4.2 OS for embedded platforms

In the embedded platforms domain, it is difficult to state the supremacy of one system over the others, since the choice depends primarily on the hardware support provided by the system, and the performance it offers in terms of memory footprint (embedded hardware resources are usually many orders of magnitude smaller then ordinary personal computers) and exported services.

One such example is TinyOS [22], which should be recognized as a de-facto standard for Wireless Sensors Networks devices. It features an easily extensible, component-based architecture and the development is based on a module-based C-dialect, called NesC. It does not offer real-time support and as such it is not suited to domains in which temporal guarantees may be required.

QNX [16] is a real-time system, tailored for embedded devices. It is a closed-source project (even if parts of it are getting released open source, as of now) and, as such, it does not allow for modifications.

LynxOS [10] is another embedded real-time system and it is very famous for is Arinc-653 [5] standard compliance (a few other OSes may offer the same benefits). Thanks to this aspect, it has been widely adopted in the medical and military fields.

For a well-written and extensive list of embedded Operating Systems, see [11].

2.4.3 Real-time operating systems

As it may result clear from the previous section, due to the intrinsic parallelism between embedded platforms and real-time applications, it often happens that an operating system designed for embedded platforms offers even real-time support.

This is obvious if we think that in most real-time domains (military, medical, industrial, ...) embedded architectures are exceptionally suited to them, since they offer superior performances in terms of mobility, weigth and dedicated hardware.

Hence, besides the previously discussed operating systems, here we will briefly list some general purpose operating systems together with another few embedded examples.

VxWorks [21] is undoubtfully the most widely adopted RTOS in the embedded domain. It features POSIX support, so UNIX programmers feel confortable with its interface. As in the case of LynxOS, VxWorks is conformant to the Arinc-653 standard, as well. Along with a suite of complete and powerful development tools (IDE and toolchains), this makes it a complete and highly mature platform for RT development.

In the Unix-like domain, a succesful approach consists of providing the system with an additional layer, with the same role as of an hypervisor, between the hardware and the OS. Probably, the three most important projects in this area are RT-Linux [18], RTAI [17] and Xenomai [25].

All of them share a common approach, that is to introduce a layer between the OS and the hardware, with the aim to totally separate the non-real time

processes execution environment from the real-time one. This idea is enforced by intercepting all the interrupts: interrupts needed for deterministic computation are rerouted towards the real-time core, while other interrupts are forwarded to the non-real time operating system which runs at a higher level (and lower priority) than during working on real hardware.

RTLinux consists of a small real-time kernel running alongside an unmodified version of the linux kernel which runs at the lowest priority. By using this small virtualization layer, the rt-kernel makes Linux a fully preemptable system.

RTAI adopts a similar approach to RTLinux, introducing, by means of the Adeos patch [2], an hardware abstraction layer on top of which a slightly modified version of Linux runs. IRQs are intercepted at lowest level by the HAL so that it can be seen as an interrupt dispatcher. RTAI considers Linux as a background task running when no real time activity occurs.

Xenomai overtook RTAI under many point of views; in particular it provides a much cleaner and more elegant code structure and interface. It suffers from a slightly higher worst-case latency when comparing IRQ dispatching and syscalls, but this is negligible with respect to the advantages that come as far as tracking problems and enriching the code with ports to new architectures are concerned.

All of these approaches are affected by the same drawback, that is a highly invasive approach with respect to the original OS. This may lead to higher difficulties in bugs scouting and fixing, as well as to more complex software platforms to maintain. In facts, the operating system core not running directly on the hardware may lead to the necessity of rewriting device drivers in order to make them aware of one more level of indirection (see, for example, the Real-Time Drivers Model [20]).

Another important approach to provide Unix-like systems (Linux, in particular) with Real-Time properties has been carried out by Red Hat and Timesys staff (Ingo Molnar, Thomas Gleixner et al.) since 2004, providing the community with the RT-Tree [8], consisting of about 1.5 MByte of patches against the

vanilla kernel (the latest release is for the 2.6.24 version) with the goal to make the Linux kernel behave according to hard real-time requirements. The patch-set consists of several improvements over the previous kernel versions, in particular in the following areas:

- in-kernel locking primitives become preemptive through the use of rt-mutexes which are also priority inheritance (we will see what it means) compliant;
- standard kernel spinlocks (another synchronization primitive, highly used in operating systems' context) are now implemented through sleeping techniques making their protected critical sections preemptible;
- a far higher precise timer infrastructure, based on hr-timers, with support for dynamic ticks has been introduced, that make the system more responsive;
- the former interrupt context is converted into preemptible kernel threads, giving the possibility to move the bottom half part of the handler code into the kernel thread;
- a high number of rescheduling points has been introduced in order to further reduce non-preemptible kernel code sections.

All of these modifications have added a better timing support and finer-grained temporal resolution, but the kernel has no notion of time, when it comes to describe task properties, but the concept of time slice, that is the dynamically updated amount of time which a task may run for before the scheduler deallocates the CPU from it.

Furthermore, with the current state of the source code, only a fixed priority (exactly like the Rate Monotonic) scheduling policy may be taken advantage of. As previously said, this can result in the CPU not being fully exploited, thus wasting system resources.

In [53] a different approach has been taken in the context of Linux. A patch introduced at kernel level exports some hooks based on which a software layer, built immediately upon it, offers to the user several real-time mechanisms and algorithms. The whole architecture has several innovative features thanks to which

AQuOSA is one of the most complete and advanced real-time platforms available to the user. Currently, the possibility of scheduling and assigning resources different from the CPU to the requesting processes is under investigation, as described in [54].

2.5 Conclusions

In this chapter we have had a brief survey on the main operating systems concepts and we have seen how these OSes relate to the classic real-time theory. By the end of the chapter we have realized that the fixed priority policy is the only choice commonly made in most notable contexts. As said, this leads, above all, to wasted CPU cycles.

We will see in the following chapters how to prevent this wasting from happening through the use of EDF-based Resource Reservations.

Chapter 3

Resource Reservations

In this chapter we are going to introduce and analyze an important class of real-time scheduling algorithms, namely *Resource Reservations* (RRES).

3.1 A brief introduction to Resource Reservations

Firstly introduced in 1994 [50], resource reservations are a framework of scheduling algorithms useful in real-time scheduling environments. It is based on the concept of isolating a scheduling resource (i.e. the CPU) when coping with a process with respect to the other ones in order to prevent a misbehaving task from affecting other tasks' execution.

This isolation (also known as *temporal protection*) is a very important property in a real-time operating system. Since on uniprocessor system only a task at a time can be in the running state, a taskset schedulability can easily be jeopardized if the system does not enforce any proper safety procedures to be deployed in case of emergency conditions.

Several experiments have been conducted in this field, especially in case of *overloaded* systems: these are systems which currently have a total utilization greater than the CPU full capacity. The way these systems cope with overrunning tasks make them more or less suitable to mission-critical environments in which a single failure compromises the whole system.

One important step through this problem has been made by Buttazzo and Abeni, with their elastic scheduling approach [31]: overrunning situations are solved with a task model which mimics the behaviour of a spring. In case of an overrun the system modifies the other tasks' parameters in order to allow for the overrunning condition to be properly handled.

Other strategies can be based of some feedback loop-based solutions, installed in the system: whenever a task is trying to overcome the amount of computation it declared during the admission test, the system deschedules it and permanently changes the parameters of the task. Possibly, the task gets discarded by the system from that point on.

All these solutions present the very same problem: they are highly invasive with respect to the original taskset specifications. Resouce Reservations are a far smoother solution to deal with these emergency conditions.

3.1.1 The need for temporal protection

As previously said, the concept of temporal protection comes to solve sudden and unexpected problems in the context of real-time scheduling. If a process does not honour its requirements in terms of execution time, it may compromise the whole taskset schedulability.

Task	C	D
A	2	3
B	1	4
C	1	12

Figure 3.1: Sample task set used in this section

In Figure 3.2 there is an EDF schedule of the taskset in Table 3.1 in which one of the tasks, due to some abrupt conditions, monopolizes the CPU for a longer time than expected, causing the so-called *domino* effect: this is a typical problem of the EDF scheduler which, trying to recover from the overrun by keeping on enforcing its policy, makes all the other tasks miss their deadlines, in sequence.

In the following section we will see how resource reservations are able to solve this problem.

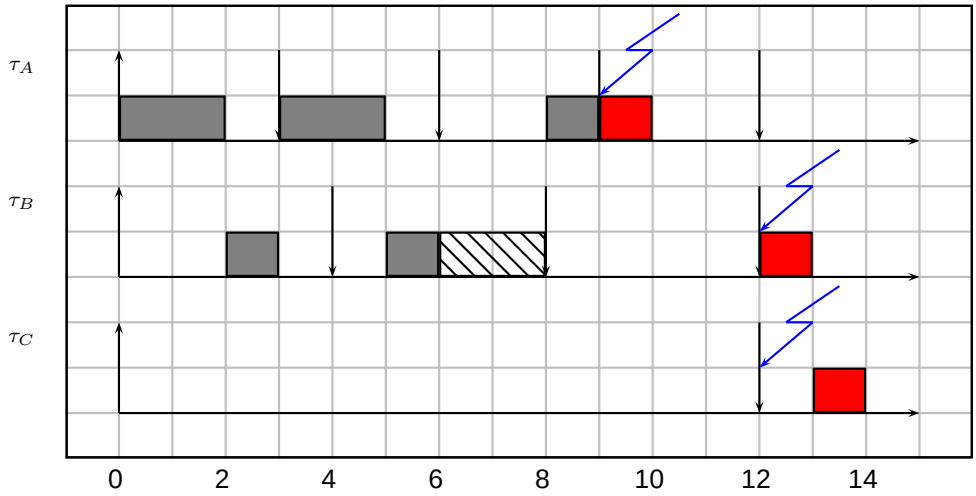

Figure 3.2: The so-called domino effect may happen in case of task overruns

3.1.2 A general definition

Resource reservations can be thought of as system *virtual* resources: as such, they represent a *fraction* of a system resource (for the purpose of this dissertation we will refer to the processor only, even though this approach is general and may be used to deal with other resource types).

A virtual resource is generally described through the concept of *Server*. A server is identified through a *budget* Q and a *period* P with the following meaning:

budget : this is the amount of resource units the server may grant to a task which is in need of service;

period : this is the server *granularity*, that is the period of time which the budget makes sense within.

As far as the CPU is concerned, a server budget represents the CPU time allotted to a task in every period. The CPU scheduling domain has been deeply studied and many RRES algorithms have been conceived, with features of very different nature.

3.1.2.1 General properties

Temporal isolation By enforcing the golden rule of *no more than Q every P*, a server may be seen as a container for its served task. As such, it works ensuring the minimum allocated bandwidth to the task's execution. At the same time, it protects the external world from possible misbehaviours from its task.

This property is of paramount importance within platforms that are sensitive to QoS degradations like video or audio applications: in such systems, the rest of the applications keep on working in the way the user expects them to, whereas the misbehaving task is the only application which would suffer from an even more delayed completion time or positive tardiness.

Real-Time guarantees Another interesting point in using resource reservations algorithms is that, by correctly tailoring the virtual resources, it is possible to maintain the very same guarantees the original task had to meet, in the context of an EDF-based schedule. In particular, the CBS behaves as a plain EDF if its parameters are chosen as follows:

- the budget Q greater or equal to the task execution time;
- the period P equal the task period.

In this way the scheduling properties are preserved.

3.1.3 Other Models

The resource reservation model, as said before, consists of allocating a share of a system resource to a requiring process. This allocation mimics a virtual private less powerful resource servicing the process. In this sense, it is possible to describe this service in different ways with respect to the budget/period metrics just presented.

3.1.3.1 The α/Δ model

In 2003, Lipari-Bini presented an alternative model [44] to describe the service provisioning in a Resource Reservation scheduling environment. In this model, given a task T_k, we define α and Δ as follows:

α_k is the computing capacity of the virtual resource which the task T_k is assigned to;

Δ_k is the maximum release delay that all the jobs of task T_k may experience without missing any deadline (represented in Figure 3.3).

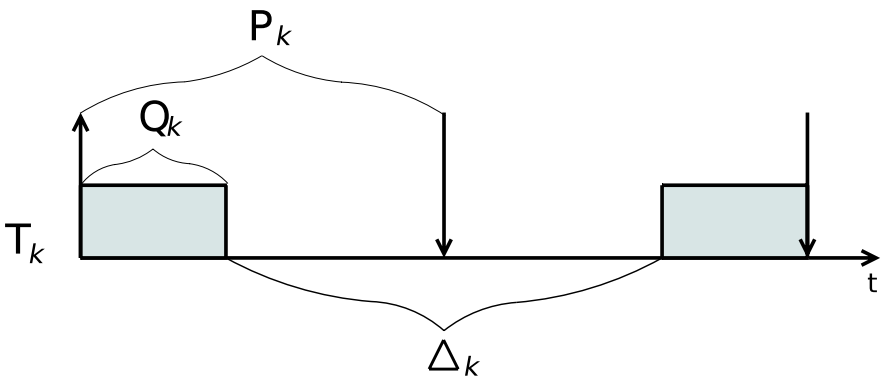

Figure 3.3: A graphical representation of the Δ_k parameter

These definitions may be exploited to represent the allocated share by means of a *service curve* (in Figure 3.4), as it is often done in the computer networks domain. A service curve is drawn on the X-Y plane. The angle it has with the X axis represents the bandwidth α, while the point d on the X axis after which the curve starts climbing is equal to Δ.

This model may be led back to the Q/P model as follows.

Theorem 1. *The α-Δ model is equivalent to a Q-P model in which $Q = \alpha P$ and $P = \frac{\Delta}{2(1-\alpha)}$.*

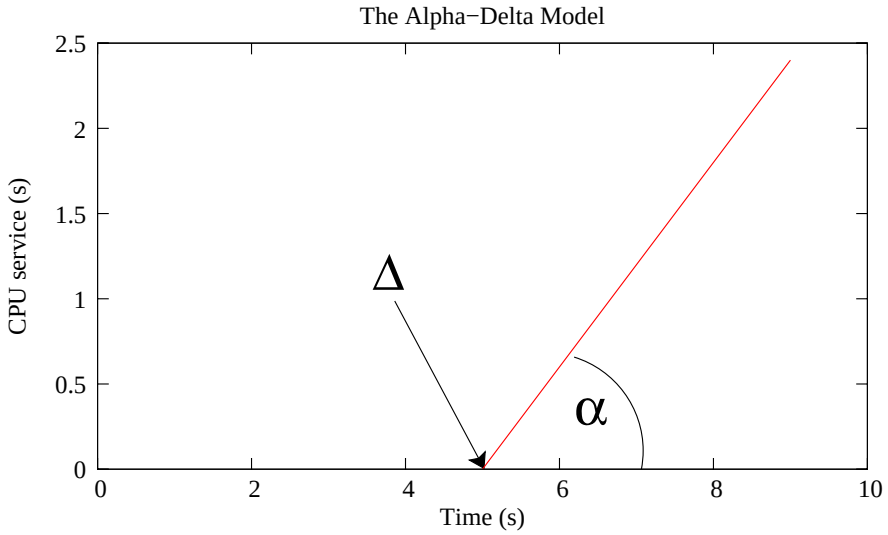

Figure 3.4: The Alpha-Delta service curve

Proof. Equivalently to the definition previously given, Δ is the maximum interval during which a task does not receive any service from its virtual resource in the context of twice the period of the task itself (see Figure 3.3). Then the following relationship holds:

$$2P = 2\alpha P + \Delta \tag{3.1}$$

Then, $\Delta = 2P(1-\alpha) \Longrightarrow P = \frac{\Delta}{2(1-\alpha)}$. □

3.1.3.2 The Virtual Time model

In 2000, Lipari proposed a new server-based algorithm [43], in order to improve the reclaiming properties of previously conceived algorithms. In this work, the idea of a server *virtual time* was presented for the first time. In this model, each server maintains an additional parameter V_t which gets updated in different ways according to the current system conditions.

In particular, every algorithm event and/or decision happens in correspondence to special virtual time values: whenever V_t increases faster, the corresponding server has less time to execute.

We will see in the following chapter how these models may be used interchangeably.

3.2 RRES algorithms survey

In this section we will briefly introduce and analyze several RRES algorithms. We will try to follow both a chronological and a logical order trying to explain what reasons reside behind the design of a new algorithm and why that one should be employed in place of another one.

3.2.1 The Constant Bandwidth Server

Conceived by Abeni and Buttazzo in 1998 [26], the CBS is the progenitor of many EDF-based resource reservations algorithms. As previously said a CBS is characterized by:

- a maximum budget Q;
- a period P;
- a current budget c;
- a deadline d;
- an associated task τ.

It works in the following way:

- each served job $\tau_{i,j}$ is assigned a dynamic deadline $d_{i,j}$ equal to the current server deadline d;
- whenever a served job $t_{i,j}$ executes, the budget q of the server S serving τ_i is decreased by the same amount;
- when $q = 0$, the server budget is recharged at its maximum value Q and a new server deadline is generated as $d_{k+1} = d_k + P$;
- a CBS is *active* at time t if there are pending jobs, otherwise it is *idle*;
- when a job $\tau_{i,j}$ arrives and the server is idle, if $q \geq (d_k - r_{i,j})U$ the server generates a new deadline $d_{k+1} = r_{i,j} + P$ and q is recharged to its maximum value Q; otherwise the job is served with the the current parameters;

In Figure 3.2.1 we see how the CBS scheduler solves the problem highlighted in Figure 3.2. The misbehaving task is isolated and it cannot affect other tasks' execution: its finishing time $f_{B,3}$ is delayed with respect to the original case, but the other tasks run unaffected and are still able to meet their deadlines.

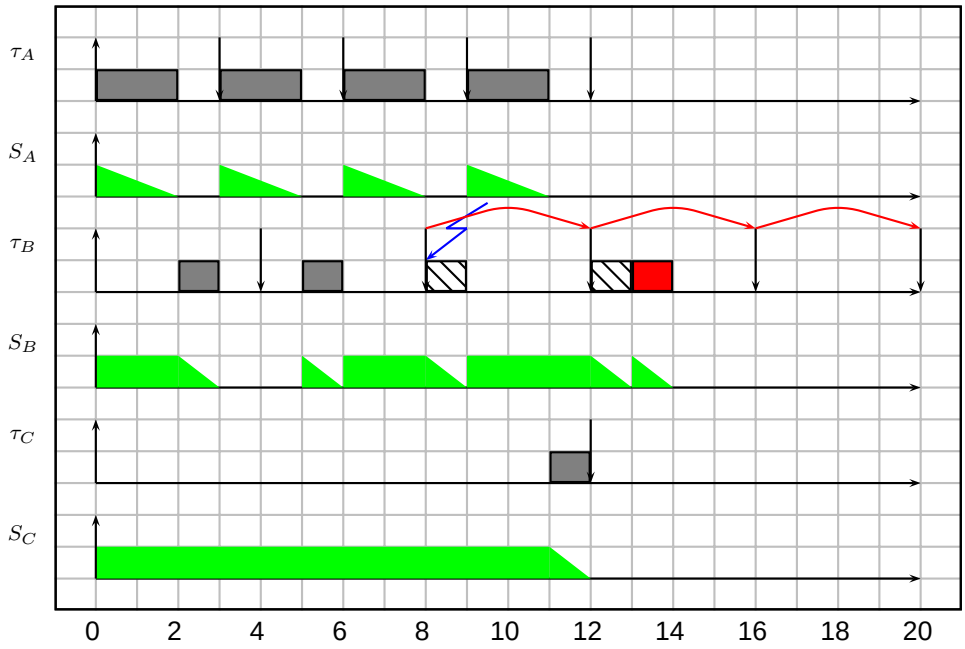

Figure 3.5: A misbehaving task is made harmless within a CBS scheduling environment

3.2.2 Greedy Reclamation of Unused Bandwidth

An important evolution of the CBS scheduling policy has been proposed in 2000 by Lipari and Baruah and is called GRUB [43]. In GRUB the virtual time model is employed. A GRUB server S_i is described by the following parameters:

- a virtual time V_{t_i};
- a period P_i;
- an utilization U_i;
- a current deadline d_i.

The algorithms keeps track of two global parameters:

- real time t;
- the current active utilization U_{act}.

Finally, GRUB works as follows:

- when a new job $J_{i,j}$ is released, $d_{i,j} = t + P_i$;
- during $J_{i,j}$ execution, $\frac{dV_{t_i}}{dt} = \frac{U_{act}}{U_i}$
- if and when $V_{t_i} = d_{i,j}$ a budget exhaustion event is thrown.

GRUB introduces a stateful description of the algorithm evolution. The states which a server may enter into are:

IDLE when there is no job released;

ACTIVE-CONTENDING when a job has been released and waits for service;

RUNNING when a job is executing;

ACTIVE-NON-CONTENDING when a job has terminated its run, but $V_t \neq d_t$. From this state, a server goes into the IDLE state later on, at a time $\bar{t} = V_t$.

All GRUB reclaiming features reside in the way the virtual time is updated. As previously described, if the system has only one server ready to run, $U_{act} = U_i$ and then V_t increases at the real time rate. This means that all the CPU is fully allocated to the only *backlogged*(i.e. which has at least a job released) server in the system. In contrast, if $\sum_{i=1}^{n} U_i = 1$, that is the system is fully loaded, V_t will increase at a rate equal to the inverse of its bandwidth, thus possibly at a much faster rate than the real time, so no reclaiming is carried out.

It is easy (and out of the scope of this thesis) to show that this reclaiming policy leads to fewer preemptions and to a fairer allocation scheme of unused bandwidth (even though it can be shown that under specific circumstances this algorithm does not perform well and ends up making some processes starve).

3.2.3 The Idle-time Reclaiming Improved Server

One of the most severe problems of CBS is depicted in Figure 3.2.3. Suppose that τ_A and τ_B are two computationally intensive processes and that at the beginning only τ_A exists. Since the CBS scheduler is work-conserving, no idle time can occur, so τ_A gets its budget (the budget of its server) immediately recharged and its deadline postponed by one period. At time $t = 7$, τ_b is ready and, since τ_A's deadline has been postponed in the future several times, τ_B will have many occasions to run, actually preventing τ_A from running for a long time. This problem is known as *deadline aging*.

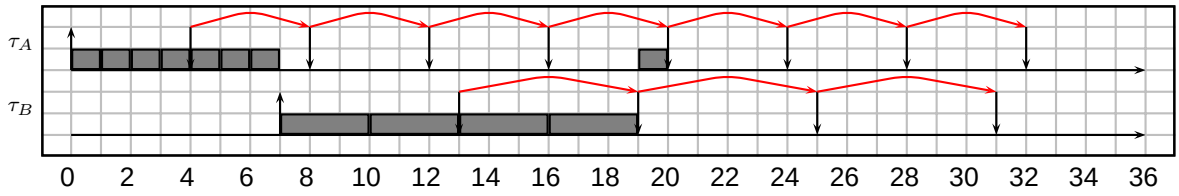

Figure 3.6: The deadline aging problem

Another more subtle problem of the CBS algorithm is highlighted in Figure 3.2.3. This simple taskset, as in the previous case, is comprised of two acyclic tasks. τ_A is the only task ready to run at $t = 0$. After a certain number of consecutive running frames from τ_A, τ_B enters the ready state and gets scheduled. The very special nature and parameters of the two tasks make the first server behave as if it had both its budget and period equal to four times the original values.

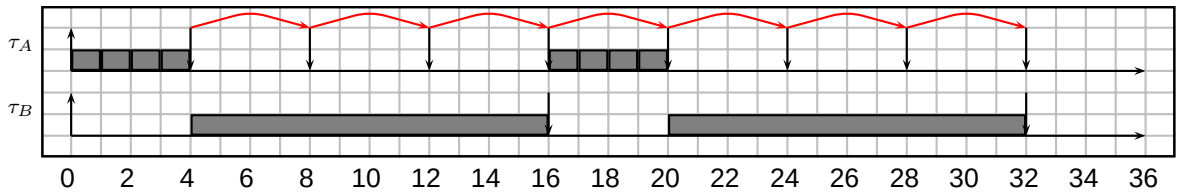

Figure 3.7: The CBS might provide a different service then what expected

This is obviously not a desirable feature, since it has a great impact on the user perception of multimedia and, more in general, delay-sensitive applications.

The IRIS algorithm [48] was conceived to solve these problems with the introduction of better reclaiming properties. It is based on the CBS scheduling policy, with the addition of two more rules:

- a hard reservation mode is enforced, so when a task is ready to run, but its server has exhausted its budget, it must wait for its current deadline before getting recharged;
- whenever there is no task ready to run, if the recharging queue is not empty all the servers' recharging times are decreased of an amount equal to $d_{i,j} - t$, where $d_{i,j}$ is the first server recharging time; this has the immediate effect to switch the first server in the recharging queue back to the ready one.

Thanks to the second rule (also known as *time warping*), the IRIS server is still work conserving, but avoiding the deadline postponing event, it ensures at the same time that in case of full CPU load no more than Q units each P are granted to the served task.

In Figure 3.2.3 we see how IRIS solves the deadline aging problem:

- each arrow represents a backward deadline postponement (it is a normal deadline moved backwards according to the second IRIS special rule);
- this is the typical behaviour of the IRIS algorithm which mimics the CBS work conserving feature;
- since the hard reservation mode is in place, whenever it tries to prevent a task from running continuosly, it triggers the time warping rule, thus not allowing deadlines to get far in the future.

In Figure 3.2.3 we see how IRIS always provides the actual service the processes requested for. The shown schedule is a direct consequence of the hard-reservation mode which, as already explained, prevents more than Q units from being allocated for each P. Here, no time warping occurs, since the CPU works at full rate.

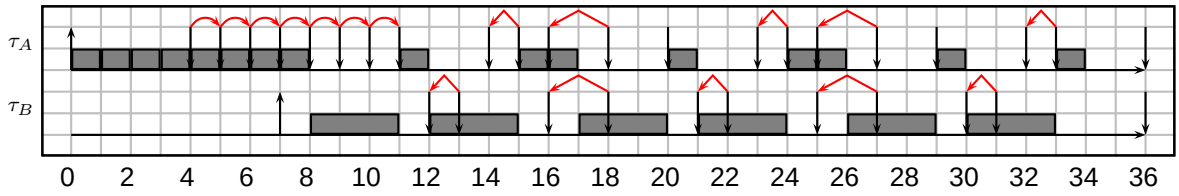

Figure 3.8: This is how IRIS solves the deadline aging problem

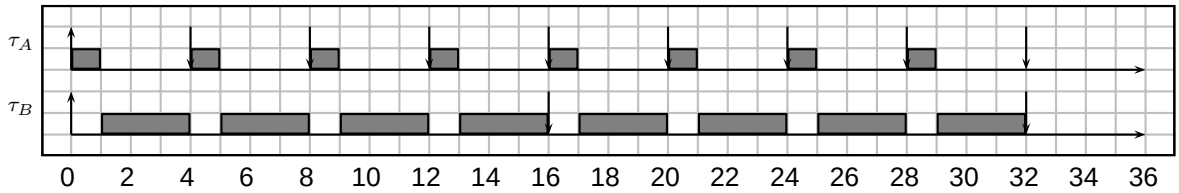

Figure 3.9: IRIS provides exactly the service requested

3.2.4 The CApacity SHaring Server

Both CBS (in soft reservation mode) and IRIS (thanks to the time warping rule) have a certain level of resource reclaiming capabilities.

Through the resource reclaiming property, the system grants no share of the reserved resource can get wasted. Altering the features of this property it is possible to award or penalize certain processes with respect to other ones according to their behaviour. As an example, the CBS reclaiming property is such that the processes with a small deadline get many more chance to reclaim for spare bandwidth than ones with long-placed deadlines. As far as IRIS is concerned a similar reasoning may be carried out.

In 2000, Caccamo et al. proposed the CASH [33] algorithm, a CBS-based RRES server with better reclaiming properties. The reasoning behind this kind of reclaiming is based on the assumption that, when jobs finish their computation leaving a current budget greater than 0, that budget may be exploited in the context of other servers (i.e. it is still valid). Whenever a budget c_i is used from a server S_j, this budget gets consummed exactly as though it were S_j's own budget, with just one difference: whenever $t > d_i$, c_i vanishes. That is, a borrowed budget

may be used until the current deadline of the server which the borrowed budget belongs to. If there is no server to exploit this budget (the processor is idle) the queued budget with the earliest deadline gets consummed exactly as though it were used by a real server.

By doing so, it can be proven that the admission test results are not compromised.

For an example to make sense it is fundamental not to exactly allocate the virtual resource parameters with respect to the original task ones, in order to let the reclaiming mechanism work. Let us consider the taskset in Table 3.10.

Task	**C**	**D**
A	5	9
B	1	4
C	1	12

Figure 3.10: The modified taskset for the CASH example

Then let us choose a set of virtual resource like the one in Table 3.11: the resulting CASH schedule is the one shown in Figure 3.12.

Server	**C**	**D**
S_A	2	3
S_B	1	4
S_C	1	12

Figure 3.11: The set of Virtual Resources serving the previous taskset

In the example of Figure 3.12 there is an additional timeline representing the bandwidth deposit. As an example, at time instant $t = 7$ the first server has a residual budget $c = 1$ and its task stops running. This means that one more budget unit may be saved for later use: this is represented in the CASH timeline with the vertical line. At the same time the second server is ready and its task starts executing. Since the CASH algorithms mandates to first utilize the saved bandwidth and then the server own one, this task executes using the CASH bandwidth (the diagonal decreasing line). But, again, it stops executing and the server unit of budget is saved (vertical again).

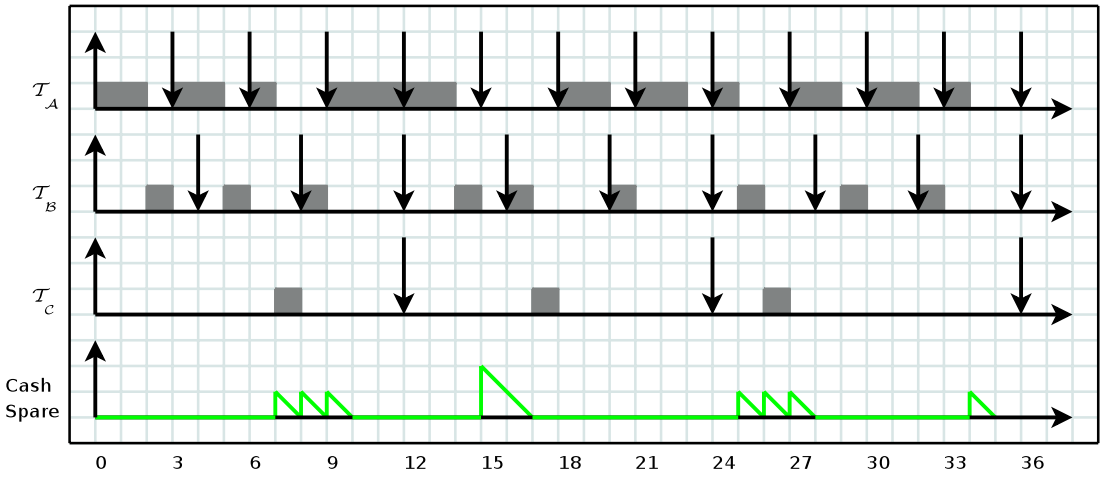

Figure 3.12: The resulting CASH schedule

Another important time instant is in $t = 15$ when the first server saves two budget units: since at that time there is no task ready to exploit it, this budget is decreased exactly as though there were a task executing.

It is interesting to notice that this mechanism may be also adopted to avoid or to better handle possible tasks' overruns: whenever there is still budget left in the CASH reserve and a task wants to execute more than declared, it may do so exploiting the additional global budget without compromising the global system schedulability (remember that the global CASH budget decreases even though there is no one to exploit it).

3.3 Picking up the right algorithm

In the previous section we had a survey on some of the main RRES algorithms employed in the context of real-time systems. The list is far from being complete, but it is useful to understand that there is a huge number of these servers out there and it is not clear on the basis of which criteria one should choose an algorithm rather than another.

In this section we will try to propose a critical comparison among several algorithms belonging to the class of Resource Reservations, through a list of

independent parameters according to which these algorithms can be partitioned:

scheduling policy, that is the underlying classic real-time scheduling policy, according to which the virtual resource takes its decisions;

reservation mode, where a hard mode means that, no matters what algorithm is chosen, no more than Q resource units will be granted every P, while a soft mode means that Q units is the minimum amount of resource units allocated every P;

reclaiming, that is the possibility for the existing servers to have their parameters stretched until the system is fully exploited; another subpartitioning concerns the way this spare bandwidth is allocated to the requiring processes (i.e. fair vs greedy).

In Figure 3.13 we propose a classification based on the set of parameters we wrote about so far. This list is far from being complete: many more algorithms exist and behave in even more complex ways, but as this dissertation has a different focus we will not go into further details.

As it is clear from the table, CBS, GRUB and CASH belong to the class of the Soft Reservation based algorithms, while CBS-HR, IRIS and BEBS provide an hard reservation based approach.

Finally, among the dynamic priorities algorithms, as far as the reclaiming properties are concerned, the only non-work conserving server is CBS-HR. All the other algorithms allocate all the spare bandwidth to the running or ready-to-run servers. A more detailed analysis must be carried out about the nature of this spare bandwidth.

3.3.1 Spare bandwidth

In a real system, it is really hard to reach the full CPU utilization through a static allocation of the available resources at design time: many other system activities must be taken into account. Unallocated bandwidth may be of different types:

1. unallocated bandwidth at admission test time;

	Scheduling policy		Reservation Mode		Reclaiming Caps	
	FP	DP	HR	SR	Yes	No
PS	X		X			X
SS	X		X			X
TBS		X	X			X
CBS		X		X	X	
CBS-HR		X	X			X
IRIS		X	X		X	
CASH		X	X		X	
GRUB		X		X	X	
BEBS		X	X		X	
SLASH		X	X		X	
BACKSLASH		X	X		X	

Figure 3.13: Main features of some important RRES algorithms.

2. non-backlogged admitted servers;

3. terminated servers' windows with a non-null remaining budget.

Spare bandwidth of the first kind is the most useful and the less dangerous to use. This is bandwidth that would be allocated in case of requests for new servers creation and, as such, may be redistributed with no risks for system schedulability.

The second type of free bandwidth concerns servers which got considered at admission time in the system, so that they hold a share of the total CPU bandwidth busy. But since they are actually not exploiting it, it may be shared among ready-to-run and running servers, with no further tests.

Finally, those servers whose bandwidth had been overestimated in first place may experience an early end of their served task instances and thus weight over the system for more than necessary. In these cases, the intuition would suggest to use the remaining budget of these servers in the context of other servers (and this is what CASH does), but this can not be done without further verifications. In fact, the algorithms seen so far consider as their threshold the instant $\bar{t}$ s.t. $c = (d - \bar{t})\frac{Q}{P}$ as the one which to base their assumptions on: in particular,

when a server is idled and has still part of its budget, this budget is valid under some specific circumstances. In the context of CASH this budget is always valid, because if no one is using it, the scheduler consummes it with no real allocation. In such a way, the CBS test of Equation 3.2 is always negative. As far as GRUB is concerned, the parameter U_{act} does not get immediately decreased, but rather at a time t such that the test of Equation 3.3 is positive.

3.3.2 Applying the models

We think it is important now to briefly reason on the way these algorithms work. Many of these servers use the very same condition used by the CBS whenever a new task instance becomes ready to run in order to decide whether the old server parameters may be exploited or not. This test is often written as follows:

$$c \geq (d - t)\frac{Q}{P} \tag{3.2}$$

If we explicit the time in Equation 3.2 we obtain:

$$\bar{t} = d - \frac{c}{U} \tag{3.3}$$

which represents the last time instant, depending on c, by which the current server parameters may be exploited. Once this threshold is passed by, new parameters have to be generated.

The CBS algorithm does not totally exploit the current parameters. In fact, whenever the test of Equation 3.2 is positive this does not mean that the current parameters must absolutely be discarded. Let us define:

$$\bar{c} = \lfloor (d - t)\frac{Q}{P} \rfloor \tag{3.4}$$

then, whenever the result of Equation 3.2 is positive, it would be sufficient to switch to $\bar{c}$ and keep the current deadline untouched. By doing so, we are *realigning* the current service curve to the ideal service modelled by the initial reservation parameters.

Another way to interpret the CBS test is by means of the virtual time. Let us further analyze this parameter and its meaning in this context. We already said

that in place of the budget Q_i and the period P_i, a server S_i can be described through an utilization U_i and a virtual time V_{t_i}. In this case the original CBS test can be reworked as follows:

$$c < (d - t)\frac{Q}{P} \Longrightarrow V_{t_i} > t \tag{3.5}$$

as the normal working condition, and as:

$$c > (d - t)\frac{Q}{P} \Longrightarrow V_{t_i} < t \tag{3.6}$$

as the condition at which new parameters must be generated.

So, at each instant, there are two time flows, the virtual and the real time ones. In standard working conditions the virtual time increasing speed stands between the real time one and the inverse of its own bandwidth; when the job stops executing, the virtual time stops and the real time may reach and overcome it. As in the previous case, though, it is possible to come up to a trade-off and partially utilize the old parameters even in case Equation 3.6 is true. It would be simply sufficient to let $V_{t_i} = t$ and let it execute unless the budget exhaustion condition, that is $V_{t_i} = d$.

As a sidenote, it is important to notice that this partial exploitation of the residual budget is not the standard policy enforced by most of the previously described algorithms, since it imposes a pretty much higher overhead on the scheduling system, due to additional context switches (reusing the current budget, whereas it would be necessary to generate new parameters, might mean two additional context switches, according to the requirements of the starting job).

3.3.3 Drawing conclusions

As a matter of fact, picking up the right algorithm in every situation is a really hard problem. Nonetheless, we deem that a good choice can be made on the basis of simple reasonings.

- Whenever a few, perfectly periodical activities exist, a well calibrated hard-reservation based server may often be more suitable than a complex reclaiming algorithm;

- when, in contrast, it is not easy to deduct the correct parameters for the reservation, it is more reasonable to exploit the reclaiming capabilities of some algorithms, in order to avoid wasting resources;
- when the processes are computationally intensive, it may make more sense to limit their possibility to monopolize the CPU by continuosly triggering an immediate budget recharging event: again, a hard-reservation mode may be more suitable;
- when the user starts highly interactive processes, it is of paramount importance that they always have a non-null current budget in order to have more chances to execute and result highly responsive.

The previous ones are very simple guidelines which may be followed when the system designer has to choose the most proper algorithm. Many other aspects can be taken into account, which could better address this choice.

3.4 Final remarks

In this chapter we have proposed a survey on the main resource reservations alternative algorithms and some criteria to take into consideration when it comes for the system designer to choose the right one.

We will see in the following chapters how to integrate this special class of algorithms within existing general purpose operating systems without imposing an excessive overhead over the system scheduler.

Chapter 4

Implementing Resource Reservations in modern Operating Systems

This chapter describes the most important practical and programming aspects of introducing real-time paradigms within general purpose operating systems. We will first analyze some important features, then we will see how to introduce a real-time scheduler inside the system. Finally we will study the possibility of introducing a resource reservations framework within these environments.

4.1 OS Real-Time compliance

In this section we are going to analyze the status of modern operating systems with respect to the possibility of providing an effective real-time support.

4.1.1 Operating systems concept of time

In the previous chapter we illustrated the class of Resource Reservation real-time scheduling algorithm. As such, it makes sense in a deadline-aware operating system only: most operating systems do not have such a notion. Being usually able to provide a time-sharing service, the concept of time flowing is mainly inherent to the time-quanta each process may exploit to execute. The problem of

how to assign these quanta to the processes is generally solved through the use of some heuristics based on the nature of the process being examined (i.e. typically cpu-bound VS interactive).

Furthermore, the concept of time flow is actually represented through the periodic increasing of a system-wide parameter whose value is updated on system timer interrupts occurrence. This is evidently a strong limitation, for it causes a continuos variable to be discretized, with all the consequent rounding errors that come from this approximation. We are not proposing anything new in this domain (new tickless operating systems implementations are currently under investigation [9]).

4.1.2 Real-time priorities

Since Unix and Unix-like systems birth, the most widely adopted solution for the process scheduler has always been to pick up a task according to its priority (statically fixed or dinamically updated). This policy has soon been extended towards the idea of a fixed priority real-time scheduler.

In modern Linux kernels, there are 100 priorities devoted to real-time scheduling according to the *SCHED_FIFO* or *SCHED_RR* policies, with the first one leaving up to the tasks to relinquish the CPU when they are done, while the second puts in action a real round-robin among the ready tasks at the same priority, exploting the very same concept of time-slice valid for non-real-time priorities.

Needless to say, even though these priorities are defined as real-time ones, the Linux vanilla kernel is far from being suitable for real-time environments, since it is not free from typical problems of general purpose operating systems that prevent them from correctly enforcing a correct scheduling sequence.

We will see in the following sections how to solve this kind of problems.

4.2 Linux

In the last few Linux kernel releases, the scheduling subsystem has undergone a rather important re-engineering process, resulting in a completely modular structure giving the possibility to write new scheduling algorithms and plug them in

the kernel as *scheduling classes*. Furthermore, a new standard scheduler, the Completely Fair Scheduler [13], has been proposed and implemented as one of the core scheduling modules. By using this module, the system is able to allocate the CPU to the requesting processes according to a fair share criterion. A fair allocation scheme, though, is usually not able to provide any kind of temporal guarantees, unless the weights are properly assigned at design time in order to meet the specific application requirements: this is not a viable approach, since the necessary computational effort increases dramatically.

4.2.1 Linux scheduling framework

Linux runs, except for `SCHED_SPORADIC`, a POSIX conforming scheduler with support for real-time and non real-time policies. Since Linux is a general purpose OS, the non real-time policy `SCHED_OTHER` is by far the most used. The code has quite recently been reworked, and turned into the so-called *Modular Scheduler Framework*, as well as provided with the group scheduling capability. Both these features are briefly described in the following subsections.

4.2.1.1 Modular Scheduler Framework

Since kernel release 2.6.23 "an extensible hierarchy of scheduler modules" is in place. Each scheduling module (*scheduling class*) is implemented in a different source file. Currently, there are only two modules: the fair scheduler module in `sched_fair.c`, for `SCHED_OTHER` tasks, and the real-time module (`sched_rt.c`), for `SCHED_FIFO` and `SCHED_RR` tasks.

The module hierarchy is made up by a linked list of available classes and the scheduler picks a ready task from the run-queue of the first module that has one. The interface each class has to implement is relatively small, i.e.:

- `enqueue_task()`
- `dequeue_task()`
- `requeue_task()`
- `task_tick()`

- `check_preempt_curr()`
- `pick_next_task()`
- `put_prev_task()`

Function names are self-explanatory, so we are not going into further details due to space reasons.

Both `sched_fair.c` and `sched_rt.c` provide the core scheduler (`sched.c`) with their own implementations of each of these functions. When the core scheduler calls them, the specific implementation of the scheduling class the current task belongs to gets invoked.

4.2.1.2 Linux and Group Scheduling

Group scheduling support has been recently introduced in the Linux kernel. This means that both tasks and tasks groups exist: they are considered as *scheduling entities*. Group scheduling entities have their own run-queues.

4.2.2 Algorithm Description

Our primary goal is to implement the standard EDF scheduling policy within the Linux Kernel, so that a real-time task may specify a minimum inter-arrival time and a worst case execution time. We assume implicit deadlines (equal to the periods). However, since Linux is a general purpose operating system, running also non real-time tasks, it is of paramount importance to correctly deal with overload conditions, as well as to face the problems of task blocking and critical sections accesses.

In case of overloads, i.e., a task trying to execute more than the WCET it specified, we force the task deactivation till the beginning of its next period, with a postponed deadline. As for critical section access arbitration the following sections will give details about the protocol we are proposing.

4.2.2.1 Critical Sections

One of the main problems in designing an efficient shared resource protocol is given by the difficulties in deriving tight upper bounds on the time spent in a critical section by a task. Since we do not want to charge the user with the task of providing such upper bounds, we developed an alternative strategy that is able to efficiently solve this problem. We chose to define a task parameter h_i specifying the maximal length for which a task τ_i can execute non-preemptively without missing any deadline. Inspired by the work developed by Baruah in [30], we provide here a simple way to compute a safe upper bound on the length of such non-preemptive chunks.

Assume tasks $\tau_1, \tau_2, \ldots, \tau_n$ are indexed in increasing deadline order, with $T_i \leq T_{i+1}$. Every task $\tau_k = (C_k, T_k) \in \tau$ is characterized by a worst-case computation time C_k, a period or minimum inter-arrival time T_k, and a relative deadline equal to the task period. The utilization of a task is defined as $U_k = \frac{C_k}{T_k}$. Let T_{min} be the minimum period among all tasks, and U_{tot} be the sum of the utilizations of all tasks.

Theorem 2 ($O(n)$)**.** *A task set that is schedulable with preemptive* EDF *remains schedulable if every task τ_k executes non-preemptively for at most h_k time units, where h_k is defined as follows*[1]*, with with $h_0 = \infty$*

$$h_k = \min \left\{ h_{k-1}, \left(1 - \sum_{i=1}^{k} U_i \right) T_k \right\}. \tag{4.1}$$

Proof. The proof is by contradiction. Assume a task set τ misses a deadline when scheduled with EDF, executing every task τ_k non-preemptively for at most h_k time-units, with h_k as defined by Equation (4.1). Let t_2 be the first missed deadline. Let t_1 be the latest time before t_2 in which there is no pending task with deadline $\leq t_2$. Consider interval $[t_1, t_2]$: since at start time no task is active, the interval is correctly defined, and the processor is never idled in $[t_1, t_2]$. Due to adopted policy, at most one job with deadline $> t_2$ can execute in the considered interval: this happens if such job is executing in non-preemptive mode at time

[1]The task with smallest relative deadline can execute non-preemptively during its whole WCET. The expression $(T_1 - C_1)$ is used to simplify the recursive formulation.

t_1. Let τ_{np} be the task which such job, if any, belongs to. The demand of τ_{np} in $[t_1, t_2]$ is bounded by h_{np}. Moreover, $T_{\text{np}} > t_2 - t_1$. Every other task executing in $[t_1, t_2]$ has instead $T_i \leq (t_2 - t_1)$. Let τ_k be the task with largest period among these tasks. Then, $T_k \leq t_2 - t_1 < T_{\text{np}}$, and $h_k \geq h_{\text{np}}$.

Since there is a deadline miss, the total demand in interval $[t_1, t_2]$ must exceed the interval length:

$$h_{\text{np}} + \sum_{i=1}^{k} \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i > t_2 - t_1.$$

Using $x \geq \lfloor x \rfloor$ and $h_k \geq h_{\text{np}}$, we get

$$h_k + (t_2 - t_1) \sum_{i=1}^{k} U_i > t_2 - t_1, \tag{4.2}$$

$$h_k > (1 - \sum_{i=1}^{k} U_i)(t_2 - t_1). \tag{4.3}$$

And, since $t_2 - t_1 \geq T_k$,

$$h_k > (1 - \sum_{i=1}^{k} U_i) T_k,$$

reaching a contradiction. □

In order to avoid complex protocols to arbitrate the access to shared resources, a good programming practice is to keep the length of every critical section short [65]. If this is the case, preemptions can be disabled while a task is holding a lock, without incurring in significant schedulability penalties. Using Theorem 2, it is possible to derive upper bounds on the time for which each task may safely execute a critical section disabling preemptions. In Section 4.2.2.2 we will explain how to efficiently use such bounds.

An efficient implementation of the test of Theorem 2 has linear complexity in the number of tasks. We hereafter present a simpler corollary that can be used to derive weaker values for the available non-preemptive chunk length, with a reduced complexity.

Corollary 1 ($O(1)$)**.** *A task set that is schedulable with preemptive* EDF *remains schedulable if* every task *executes non-preemptively for at most*

$$h = (1 - U_{tot}) T_{\min}$$

time units.

The above theorem allows computing one single system-wide value h for the allowed maximum non-preemptive chunk length *of all tasks*, in a constant time. This lighter tests is a valid option whenever it is important to limit the overhead imposed on the system.

In the following sections, we will compare the solutions given by Theorem 2 and Corollary 1, in terms of schedulability performances and system overhead. Other more complex methods may be used to derive tighter values for the allowed lengths of non-preemptive chunks (see, for instance, [30]); nevertheless, we chose not to use such methods due to their larger (pseudo-polynomial) complexity. Having a fast method to calculate a global value for h is, in our opinion, really important, as in a highly dynamical system, with thousands of tasks, as Linux can be, it allows our method to be used without significant overhead.

Using a global value for h simplifies the implementation and reduces the run-time overhead of the enforcing mechanism, that has not to keep track of the per-task values.

It is worth noting that more sophisticate shared resource protocols like the Stack Resource Policy (SRP) [29] are not so suitable for the target architecture, since they are based on the concept of ceiling of a resource. To properly compute such parameter, it would be necessary to know a priori which task will lock each resource and, in a real operating system, this is definitely not a viable approach from a system design point of view.

4.2.2.2 Admission Control

One of the key points of our approach is that there is no need for the user to specify a safe upper bound on the worst-case length of each critical section, something that is very problematic in non-trivial architectures. The system will use all the available bandwidth left by the admitted tasks to serve critical sections, automatically detecting the length of each executed critical section, by means of a dedicated timer. If some task holds a lock for more than the allowed non-preemptive chunk length, it means that some deadline may be missed, and the system is overloaded. In this case, some decision should be taken to reduce the

system load. There are many possible heuristics that can be used to remove some task from the system to solve the overload condition, the choice of which depends on the particular application. For instance, the system may reject the most recently admitted tasks; or it can reject tasks with heavier utilizations or longer critical sections, leaving enough bandwidth for the admission of lighter tasks; it can penalize less critical tasks, if such information is available; or it can simply ask the user what to do. We chose to reject the task with the largest critical section length, which is the one that triggered such scheduling decision executing for more than the allowed non-preemptive chunk length.

The system keeps track of the largest critical section R_i for each task τ_i, triggering a timer at the beginning and at the end of each critical section. Since every task will execute non-preemptively while holding a lock, one single timer is sufficient.

The admission control algorithm changes depending on the complexity of the adopted method to compute the time for which a task may execute with preemptions disabled. We distinguish into two cases: (i) using for all tasks a single system-wide value h given by Corollary 1; or (ii)using for each task τ_i a different value h_i given by Theorem 2.

In the first case the system keeps track of the largest critical section among all tasks: $R_{\max} = \max_{i=1}^{n}\{R_i\}$. For all deadlines to be met, this value should always be lower than the current non-preemptive chunk length:

$$R_{\max} \leq h. \tag{4.4}$$

When a task τ_k would like to be admitted into the system, the following operations are performed:

- The allowed non-preemptive chunk length h' after the insertion of the new task is computed using Corollary 1.
- If such value h' is lower than the maximum critical section length among the already admitted tasks $R_{\max}$, the candidate task is rejected, since it means that there would not be enough space available to allocate the blocking time of some task.

- Otherwise, task τ_k is admitted into the system, updating h to h'. Note that $R_{\max}$ does not need to be updated, since there is no available estimation of the maximum critical section length of τ_k (initially, $R_k = 0$).

When a task τ_k leaves the system, the new (larger) value of h is computed and accordingly updated. Moreover, if $R_k = R_{\max}$, $R_{\max}$ may as well be updated (decreased).

In a certain sense, we can say that a task is *conditionally* admitted into the system, and it will remain so as long as it does not show any critical section that is longer than the maximum non-preemptive chunk length allowed, in which case the task is rejected from the system. Alternative strategies may instead trigger different scheduling decision when $R_{\max}$ exceeds h, for instance creating room for a task with a long critical section by rejecting different tasks.

The slightly more complex case in which different non-preemptive chunk values h_i are used for each task τ_i, we will instead proceed as follows. In order to guarantee that all deadlines be met, we will check that every task τ_i has a non-preemptive chunk length h_i sufficiently large to accommodate the maximum critical section of that task:

$$\forall i, \quad R_i \leq h_i. \tag{4.5}$$

When a task τ_k would like to be admitted into the system, the following operations are performed:

- Using Theorem 2, we compute the allowed non-preemptive chunk length h'_i after the insertion of the new task, for all tasks τ_i having a period at least as large as τ_k's: $T_i \geq T_k$.

- If there is at least one value h'_i that is lower than the maximum critical section length of the corresponding task τ_i — i.e., $h_i < R_i$ — the candidate task τ_k is rejected.

- Otherwise, τ_k is admitted into the system, updating each h_i to h'_i.

When a task τ_k leaves the system, we simply recompute the h_i values of the tasks with period greater than T_k.

Independently from the adopted strategy, the system will check if an invariant condition (given by Equation (4.4) or Equation (4.5) is maintained. When it is not, some decision should be taken to solve the overload condition.

4.2.2.3 Exported Interface

One of the largest issues we faced at design time was deciding what interface our scheduling algorithm should export.

We strive for something not too different from the existing Linux scheduler interfaces. Moreover, we want the user to be able to code both periodic and sporadic tasks, as well as both hard and soft real-time applications. Finally, we think it would be useful to implement an already existing and widely adopted interface, so to make real-time programmers as comfortable as possible with it.

For all these reasons, we looked at the Ada 2005 [1] programming language specification, since EDF dispatching is included, as briefly shown below.

Ada 2005 EDF Dispatching Interface Ada 2005 EDF dispatching package [15]:

```
with Ada.Real_Time;
with Ada.Task_Identification;
package Ada.Dispatching.EDF is
        subtype Deadline is Ada.Real_Time.Time;
        Default_Deadline : constant Deadline :=
                Ada.Real_Time.Time_Last;
procedure Set_Deadline(D : in Deadline;
        T : in ADA.Task_Identification :=
        Ada.Task_Identification.Current_Task);
procedure Delay_Until_And_Set_Deadline(
        Delay_Until_Time : in Ada.Real_Time.Time;
        Deadline_Offset :
                in Ada.Real_Time.Time_Span);
function Get_Deadline(
        T: Ada.Task_Identification.Task_ID :=
        Ada.Task_Identification.Current_Task)
        return Deadline;
end Ada.Dispatching.EDF
```

It is obvious how to exploit this interface to program either periodic or sporadic task. In particular, a call to Delay_Until_And_Set_Deadline() delays the calling task until time Delay_Until_Time. When the task becomes runnable again it will have

$$deadline = Delay\_Until\_Time + Deadline\_Offset$$

Linux EDF Interface The Linux scheduler interface is based on `sched_{set, get}scheduler` and `sched_{set, get}param` system calls, and on the `sched_setscheduler` data structure. Since we definitely want to avoid binary compatibility problems with legacy applications, we did not change neither the behaviour of these functions nor the size of the data structures involved.

Moreover, as stated before, we want something similar to the Ada 2005 interface, provided that some adaptation to the specific context(i.e., the Linux kernel) is unavoidable.

Hence, we used a new data structure for EDF scheduling parameters, `sched_param2` and we added four new system calls:

- `sched_{set, get}scheduler2`;
- `sched_{set,get}param2`.

The new `sched_param2` has room to accommodate the period and the maximum possible runtime. It also contains a `deadline` field, useful for reading the current (absolute) deadline of a task.
In particular, the new `sched_setscheduler2` system call, which takes a `sched_param2` as an argument, behaves as follows:

1. it sets the parameters passed as the new current ones for the calling task;
2. it makes the task sleep until the relative time specified in the `sched_param2` argument passed, as the period;
3. on task wake-up, it sets its deadline to the current time plus the time value specified in the `sched_param2` argument that is passed, as the deadline.

Like in Ada 2005, both the sporadic and the truly periodic task models may be described through this interface.

4.2.2.4 Implementation details

Implementation has been carried out with efficiency in mind. The queue accommodating ready-to-run tasks is kept deadline-ordered, for $O(1)$ extraction. Furthermore, we implemented it as a red-black binary tree (RB-Tree) to keep also insertion and deletion efficient with a $O(logn)$ time complexity and because the kernel exports an highly optimized RB-Tree implementation, being it used in several other subsystems.

EDF Scheduling Class Implementation Exploiting the modularity provided by the new scheduling framework, we implemented the EDF algorithm inside a new scheduling class. This means we added the file `sched_edf.c` and placed it as the head of the scheduling classes linked list, so that a ready EDF task will always have the highest priority in the system.

4.3 Experimental evaluations

To evaluate the effectiveness of our solution, we performed a series of experiments with randomly generated task sets.

We generated a great number of task sets, varying the total utilization U_{max} the task parameters T_k, C_k, and the length of the critical sections shared among the tasks. We used values of the total utilization U_{max} ranging from 0.4 to 0.9, and for each value, we evaluated different task behaviors.

We first considered values of U_k generated from an exponential distribution with $\lambda = 0.1$, with periods T_k uniformly distributed in the interval $[1, 1000]$; then we varied both λ and the range used for T_k generation. The acceptance rates for tasksets with $\lambda = 0.01$ and T_k generated from $[1, 10000]$ are shown in Figure 4.1; as expected, increasing the total utilization, the $O(1)$ test becomes less effective, while the $O(n)$ test shows a graceful degradation.

From the experiments, with the generated tasksets, we have observed that the values for h_k given by the $O(n)$ test were pretty uniform, indicating that the component given by the highest priority task often dominates the others. This is obviously positive, as it means that tasks are allowed longer non-preemptive

regions, and can be seen as the increase of T_k dominating the decrease in U_k in the $O(n)$ formulation. The h_k values' span observed in this first test is shown in Figure 4.2. The figure shows max $\left(\frac{\max_k h_k - \min_k h_k}{T_{max}}\right)$ over all the generated tasksets of a given utilization.

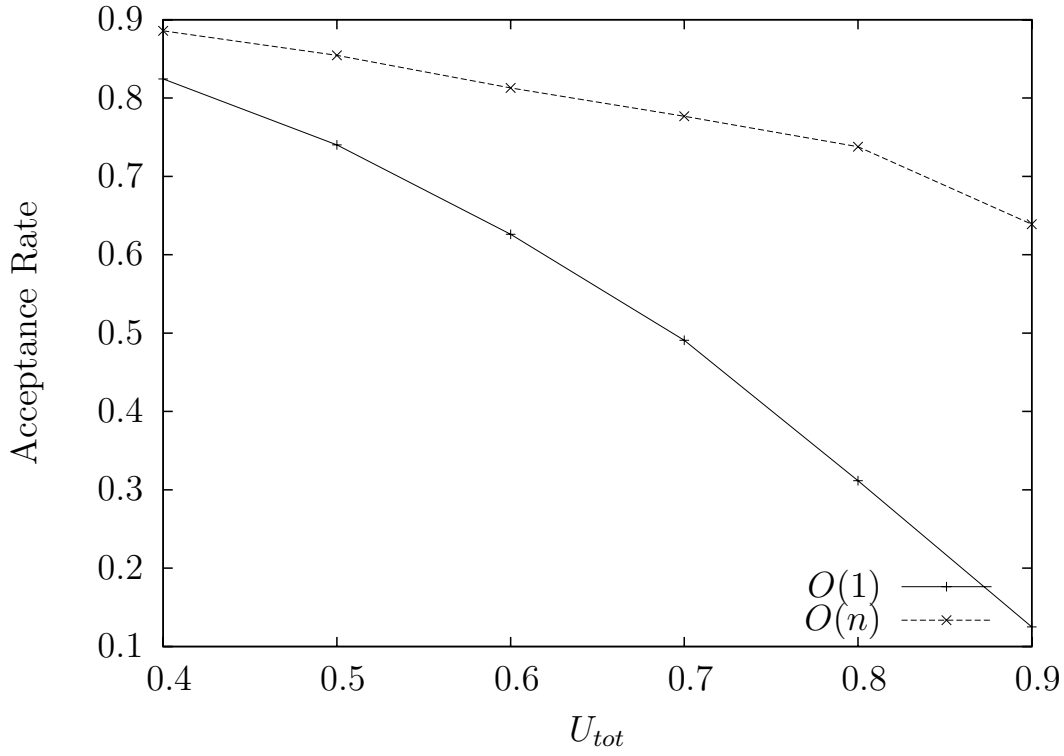

Figure 4.1: Light Tasks, Big Period Span.

4.4 MINIX 3

We now focus on a completely different operating system architecture, that is a microkernel-based operating system like MINIX 3. The need for such an architecture has recently increased, since modern computer users are always more concerned about system dependability.

While end-user requirements used to represent a trade-off between performance and costs, developers nowadays have to meet the demand for hard safety guarantees. This includes security and privacy, robustness against failures, time-

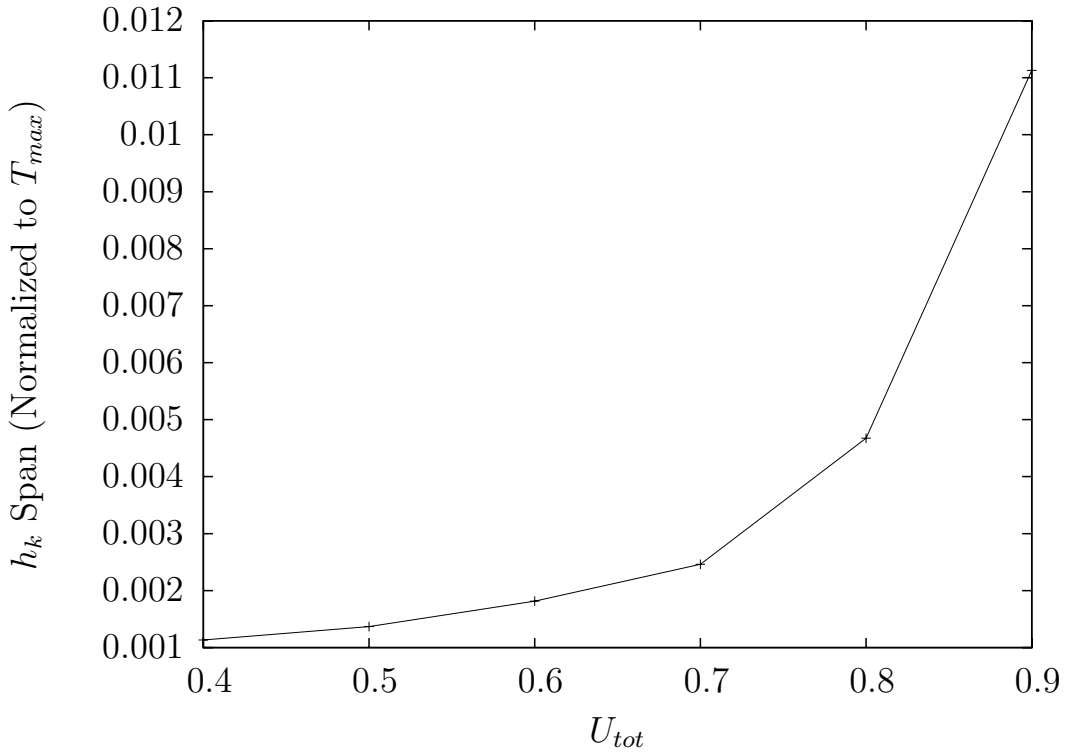

Figure 4.2: h_k Span for Light Tasks, Big Period Span.

liness of operation, quality of service, and so on. The dependability axis we refer to concerns the temporal domain.

A recent study [64] showed that common process scheduling mechanisms can be subverted in a practical manner without superuser privileges in order to monopolize the CPU. This is a threat to not only timesharing systems, but also embedded systems such as cell phones, PDAs, etc. The cheating process effectively gains the maximum priority, performing a denial of service (DoS) attack on other tasks. It was shown that almost all current operating systems, including MINIX 3 according to our analysis, are affected by this problem.

Furthermore, timeliness of operation is important in many application domains, including multimedia, VOIP, peer-to-peer services, interactive computer games and so on. Each of these domains has its own peculiarities, but all of them share an equal need of a minimum guaranteed service level. A best-effort service based on heuristic algorithms is usually adopted in order to improve the

end-user perception of the overall quality, but this approach fails to provide minimum service guarantees. For example, in an attempt to keep the highest possible throughput, the performance of certain critical services may be heavily degraded under high-load conditions, which may lead to a low quality of service as perceived by the end user.

In order to improve the every-day user experience for such time-sensitive applications, a real-time operating system (RTOS) adapts the computational resources granted to each application based on its quality-of-service requirements. To date much development has focused on adding real-time features to commodity, monolithic, PC operating systems, such as Linux [27, 40, 56]. In contrast, the work in this paper provides real-time support on MINIX 3, a novel microkernel-based, multiserver operating system. This focus allowed for a highly modular design and implementation of a reservation framework with only modest modifications to the base system.

4.4.1 OS general description

MINIX 3 is a microkernel-based multiserver operating system for uniprocessors that is designed to be extremely fault-tolerant. All system services run as highly restricted user-mode processes in order to isolate faults occurring in one component and prevent the damage from spreading, so that the rest of the system can continue to function normally. In contrast, a bug in a kernel module in a classic monolithic operating system could easily hang or crash the entire system due to the lack of isolation. In addition, the extension manager can detect certain error conditions, including failures relating to CPU or MMU exceptions, internal panics or infinite loops, and restart faulty processes. These features greatly improve the system's dependability [38, 39].

In addition to dependability, MINIX 3's highly modular structure makes it a good candidate as a real-time operating system for embedded platforms. Its code base is several orders of magnitude smaller than Linux, it is easy to remove unwanted components in order to get a minimal configuration, and the simple structure results in a small memory footprint. Moreover, MINIX 3 already has good response times due to the following design choices:

- the user-mode operating system servers and drivers have short servicing

times and are fully preemptible by higher-priority processes,

- the kernel has very short interrupt latencies because its generic interrupt handler only masks the IRQ line and sends a notification message, whereas the actual interrupt handling is done by a user-mode driver, and
- finally, the kernel has short atomic kernel calls, which results in low stuck-in-kernel latencies.

However, MINIX 3 did not yet explicitly address other real-time application requirements. Realizing real-time behavior is not straightforward, since standard MINIX 3 versions lack important real-time properties, including:

- a way to describe a task's real-time constraints and schedule it accordingly,
- a temporal profile of each component in the system in order to achieve a complete system predictability, and
- typical resource access protocols, such as Priority Inheritance [59] or Stack-Based Resource Protocol [28], in order to avoid priority inversion phenomena.

4.4.2 Resource Reservations

In order to provide temporal protection on MINIX 3, we have made several modifications to the scheduler and designed and implemented a RRES framework.

To the best of our knowledge, we are the first to implement resource reservations in MINIX 3. In particular, we have provided a complete implementation of CBS [27], CBS-HR and IRIS [48], which are among the first and most effective ones. The new resource reservation framework improves MINIX 3 in three important ways:

1. RRES brings soft real-time support, so that benefits can be gained in many application domains, like the ones mentioned above. Correct accounting is achieved under the assumption of MINIX 3's low-latency response times discussed above. Moreover, infrequent deadline misses are tolerable due to the nature of soft real-time applications; the end user will perceive a missed deadline as a quality-of-service degradation rather than a fatal error.

2. Although our primary focus is *soft* real-time support, the RRES framework also provides limited *hard* real-time support for applications that do not rely on the standard system servers and drivers, such as sensoring applications using memory-mapped I/O. The only critical code is the kernel's generic interrupt handler, which has a short, strictly bounded execution time.

3. Our work improves dependability by enabling temporally isolated execution in order to prevent denial of service attacks [64]. Reliable accounting is realized by using the TSC cycle counter independent from the programmable interrupt timer (PIT), as detailed in Sec. 4.4.4.4.

4.4.3 Related Work

We distinguish different operating system structures, since each structure leads to different real-time properties.

4.4.3.1 Monolithic Operating System Structure

In spite of significant research efforts, introducing real-time support in monolithic systems, such as Linux, is still considered an open problem. Real-time scheduling turned out to be difficult, mainly due to the presence of many other highly unpredictable system activities, such as interrupt handling, paging and process management.

Two approaches have been adopted in order to minimize latencies and improve response times. First, shortening non-preemptible kernel code sections. This changes local code sections, but keeps the same monolithic kernel structure. As an example, Red Hat staff has contributed a series of kernel *low-latency patches* to the Linux community [8]. The patches have proven to be effective and are a substantial step towards a real-time Linux.

Second, introducing an additional real-time layer between the operating system and the real hardware in order to actively handle real hardware interrupts and mask them to the operating system when needed. This results in an hybrid architecture with a monolithic kernel running on top of a microkernel layer. The most important projects are RTAI [17], RT-Linux [18] and Xenomai [25]. All these projects adopt a similar approach to the problem: a new interrupt dis-

patcher is added below the standard kernel which traps the peripheral interrupts and reroutes them to Linux whenever it is necessary. However, this approach means that real-time tasks cannot directly access standard Linux services and existing device drivers due to potentially high and unpredictable delays. For this reason, developers often have to (re)write their own real-time drivers.

4.4.3.2 Multiserver Operating System Structure

Real-time work also has been done in the context of multiserver systems. Here, low interrupt latencies and good response times are easier to achieve than in a monolithic system, since all services are already scheduled independently. Below, we discuss related work in three systems.

Resource reservations and temporal protection have been tested before on Real-Time Mach (RT-Mach) [49, 51, 62]. RT-Mach enforced the concept of resource reservation using a fixed-priority schemes like RM [46], or, at most, a dynamic-priority scheme based on old algorithms like TBS [60], which cannot achieve full CPU utilization. In contrast, MINIX 3 implements the newer CBS, CBS-HR and IRIS algorithms. Furthermore, RT-Mach seems to have fixed the scheduling policy in the kernel, whereas we promote a minimally invasive, modular design.

Real-time support in L4 [42] is based on the statistical approaches Quality-Assuring Scheduling (QAS) [36] and Quality-Rate-Monotonic Scheduling (QRMS) [37]. By extracting task properties, the system can guarantee that the deadlines of the mandatory part are met, while deadline misses in the optional part are tolerated. However, in order to enforce the mandatory-optional splitting principle, DROPS' real-time applications require modifications at source code level, whereas our framework can directly serve any existing applications in a real-time fashion. Furthermore, QAS and QRMS can provide guarantees for only periodic tasks, whereas CBS, CBS-HR and IRIS also support aperiodic tasks with real-time requirements. We also believe that our implementation can be simpler, since no complexity is introduced at admission and reservation level, whereas QAS performs these tasks using the distribution of execution times.

Finally, two projects based on earlier versions of MINIX should be mentioned. First, Minix4RT [55] aims to mimic the low-latency RT-Linux architecture in MINIX 2. Second, RT-Minix [57, 58] consists of a set of system calls added to

MINIX 2 in order to explicitly invoke real-time services provided by the kernel level. The former project has been made obsolete by MINIX 3, since its generic interrupt handler achieves low interrupt latencies in a much simpler way. Furthermore, these approaches are too invasive with respect to the base system and cannot be easily ported to MINIX 3. Moreover, our work provides the first-ever implementation of resource reservations and temporal protection based on CBS, CBS-HR and IRIS in the context of MINIX 3.

4.4.4 Design and implementation

This section describes how we implemented a resource reservation (RRES) framework in MINIX 3 with support for the CBS, CBS-HR and IRIS resource reservation algorithms. Three important design guidelines for the implementation of the RRES framework were:

1. pluggable real-time support next to best effort;
2. minimizing the amount of intrusive kernel code;
3. maximizing the policy-mechanism separation.

First, we did not want to break the standard MINIX 3 distribution for reasons of acceptance and backward compatibility. Therefore, we designed the RRES framework as an optional component that can be started at run-time to enhance the system with real-time support when needed. Second, a general dependability strategy in MINIX 3 is to move as much code as possible out of the kernel into user space. Since kernel-mode code runs with all privileges of the machine it must be fully trusted, whereas user-mode bugs may be confined to the process in which they occurred. Third, separating the scheduling policies from mechanisms leads to a flexible, easily adaptable system. Fortunately, these guidelines go hand in hand, as discussed below.

4.4.4.1 High-level Design Overview

Based on the above design criteria we decided to introduce a separate user-space scheduler, called the RRES manager or *RRES* for short, which is logically located at the MINIX 3 server level. RRES can be started through the MINIX 3 extension manager at run-time like all other extensions [39]. The basic idea then is to let

the kernel execute user-space scheduling requests for real-time applications on behalf of RRES. In particular, the kernel's built-in best-effort scheduling policies should be temporarily suspended, so that the real-time task is not affected by the heuristics of the standard scheduler. In other words, the scheduling policy is enforced in user-space, but the kernel provides mechanisms for starting and stopping a task and for accounting its execution. Logically, this leads to a separate RRES scheduler next to the MINIX 3 scheduler.

As an aside, we provided three different implementations of the RRES manager, one for each resource reservation algorithm supported: CBS, CBS-HR, and IRIS. The algorithm used is statically chosen with a compiler flag. It is currently not possible to let different VRESes serve their task using different algorithms, since the theoretical analysis to make this possible is still in progress. The reason for supporting CBS and CBS-HR next to IRIS is a matter of usability. With IRIS' time warping rule, all CPU cycles would be used for real-time task and non-real-time applications would not get a chance to execute. In such a scenario, every application should be enclosed in a reservation, resulting in a system that is harder to analyze and maintain.

In addition to the RRES manager, three helper utilities were created in order to manage real-time applications. First, *rres_create* can be used to start a new real-time application by passing the binary's name its period P and budget Q. Second, *rres_change* can be used to change the scheduling parameters at runtime. Third, the *rres_destroy* utility can be used to stop a running real-time task. Fig. 4.3 gives a high-level overview of the RRES framework.

4.4.4.2 Implementation of the RRES Manager

The RRES manager has the same code structure as other MINIX 3 servers. After the initialization of its data structures, RRES starts a never-ending loop in which it accepts new requests, processes them and sends back an answer.

RRES Data Structures The main RRES data structure has three scheduling queues for the virtual resources that are uniquely associated with the real-time tasks. The queues are ordered by increasing current VRES deadline, so that RRES can quickly decide which task to schedule based on the underlying EDF policy.

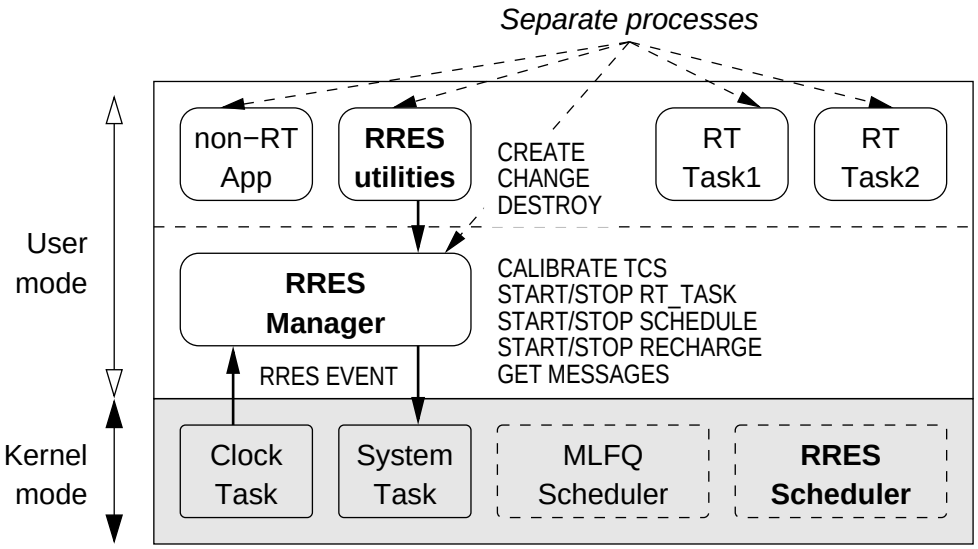

Figure 4.3: High-level architecture over the resource reservation framework. Messages exchanged between the RRES helper utilities, RRES manager and kernel are shown.

- The **ACTIVE** queue keeps track of ready-to-run VRESes. The first VRES on this queue is the currently scheduled one, that is, the associated task is the running process in the system.
- The **RECHARGING** queue comprises all the VRESes which exhausted their budget and need it to be replenished. This queue is only used for CBS-HR and IRIS. With plain CBS it is always empty since hard-reservation mode is not used. Conceptually, all VRESes in this queue are recharging, but RRES only sets a single alarm for the first recharging event.
- The **BLOCKED** queue, finally, contains the VRESes that blocked during their execution, for example, because they have to wait for some event to happen.

RRES Interactions As shown in Fig. 4.3, the RRES manager has several interactions with both the RRES help utilities and the kernel tasks. The exact messages that are exchanged are shown in Fig. 4.4. First, the RRES helper utilities can request RRES to CREATE, CHANGE or DESTROY virtual resources. In order to prevent random tasks from changing their scheduling policy only the

system administrator is allowed to send RRES requests. RRES verifies this by asking the MINIX 3 process manager for the requester's user ID.

Helper Utility -> RRES Manager	RRES Manager-> Kernel task	Kernel task -> RRES Manager
1. CREATE	1. CALIBRATE_TCS	1. RRES_EVENT
2. CHANGE	2. START/STOP_RT_TASK	– budget exhausted
3. DESTROY	3. START/STOP_SCHEDULE	– recharge time
	4. START/STOP_RECHARGE	– task blocked
	5. GET_MESSAGES	– task unblocked
		– task exited

Figure 4.4: Messages exchanged within the RRES framework.

Second, although RRES is responsible for the scheduling policy, it relies on kernel mechanisms to perform the actual RRES scheduling. In particular, the following messages are exchanged with the kernel's system task:

- CALIBRATE_TSC: used at RRES initialization time to determine the number of CPU cycles per microsecond; the kernel programs the timer to a known frequency, reads the TSC cycle counter start value, waits 1000 timer ticks, and reads the TSC end value.
- START_RT_TASK: tell that a process now is a real-time task and needs to be treated in a special manner.
- STOP_RT_TASK: inform the kernel that a real-time task has been destroyed so that special events related to this task are no longer forwarded to RRES.
- START_SCHEDULE: tell the kernel to start scheduling a real-time task using the RRES scheduler rather than the standard scheduler.
- STOP_SCHEDULE: issued whenever RRES needs to stop the currently scheduled real-time task.
- START_RECHARGE: if a VRES becomes the head of the RECHARGING queue, RRES schedules an alarm to be notified when the recharging time is reached.

- STOP_RECHARGE: used to handle a time warping event in IRIS and if the scheduling parameters of a currently recharging task are changed.
- GET_MESSAGES: whenever the kernel's mechanisms encounter a special event, as shown in Fig. 4.4, the RRES manager is notified with an RRES_EVENT message; the RRES manager then makes a callback to find out which event triggered the notification.

While this modularity brings many benefits with respect to flexibility, the message passing interactions between RRES and the kernel introduces a small latency. Experiments on a prototype implementation have shown, however, that the incurred context-switching overhead is not at all prohibitive, as discussed in Sec. 4.4.6.

4.4.4.3 Kernel and Scheduler Modifications

Scheduling in the standard MINIX 3 kernel is done on best-effort basis using a multilevel-feedback-queue scheduler (MLFQ) [63]. Processes with the same priority reside in the same queue and are scheduled round-robin. When a process is scheduled, its quantum is decreased every clock tick until it reaches zero and the scheduler gets to run again. To prevent starvation of low-priority processes, a process' priority is degraded whenever it consumes a full quantum. Since CPU-bound processes are penalized more often, interactive applications have good response times. Periodically, all process priorities are increased if not at their initial value.

As mentioned above, the kernel should bypass the standard scheduler for real-time tasks managed by RRES. Therefore, the MINIX 3 kernel and scheduler were changed in two ways. First, we added *rres_f* flag to the process structure in order to tell whether a task should be scheduled in the context of MLFQ or RRES. This flag is set when RRES sends a START_SERVE request to the kernel. Second, the scheduler data structure was extended with two new scheduling queues at the highest priorities, as shown in Fig. 4.5.

- **RRES_PRIO**: the highest priority in the system is now used for the RRES manager, so that it can always immediately react to the various kinds of events, such as budget exhaustion and budget recharged events. Depending

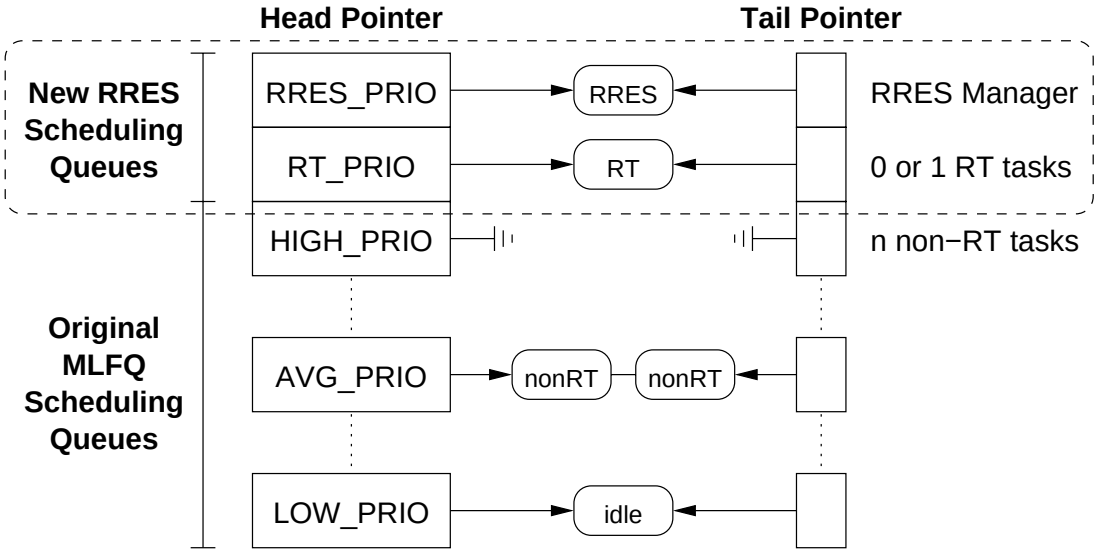

Figure 4.5: RRES-enhanced MINIX 3 scheduling queue data structure. Two new queues at the two highest priority levels were added for the RRES manager and the current real-time task.

on the kind of event RRES may schedule another real-time task. When RRES has processed the event, it returns to its main loop and blocks waiting for the next event—allowing a real-time task to run.

- **RT_PRIO**: the second highest priority is reserved for the real-time tasks served by the RRES manager. At most a single task can be active at any given time. When there is a task to schedule, it runs uninterrupted until either its budget is exhausted or some other RRES event makes a higher-priority task ready to run. In the latter case, *preemption* occurs and RRES requests the kernel to schedule the higher-priority task.

Third, we identified the points which needed change in order to modify the default scheduler behavior. In particular, if a real-time task needs to be scheduled, that is, if a process' *rres_f* flag is set, the scheduler simply picks the queue with priority level RT_PRIO rather than its MLFQ priority. Also, a task running in the RT_PRIO queue is not affected by the heuristics of the normal MLFQ algorithm, such as decreasing the process priority of long-running processes and periodic balancing of the scheduling queues.

Finally, we changed the scheduler to cope with blocking and unblocking events.

Whenever a real-time task blocks the kernel sends an event notification to RRES, so that it can schedule another task. Blocking can occur, for example, during synchronous service requests or while waiting for an I/O completion interrupt. We decided to consider a task's blocking and unblocking events as job completion and activation times respectively in order to be able to provide the classic real-time properties previously described. The blocked task's VRES is put on RRES' BLOCKED queue. When the kernel notifies RRES that the task is unblocked, RRES moves the corresponding VRES to the ACTIVE queue and may schedule it depending on its current priority.

4.4.4.4 CPU Time Accounting

In order to serve real-time tasks the RRES framework requires a reliable source of high-precision timing. Our implementation is based on the x86's TSC cycle counter, but depending on the system architecture, other timing sources may also be available. The TSC cycle counter is convenient because it is accessible to both the user-space RRES manager and the kernel's scheduling code. However, since the TSC cycle counter is read-only and cannot interrupt when a task's budget is exhausted or needs to be replenished, an interrupt-based programmable timer is also needed. For this, we decided to modify the standard MINIX 3 system timer, which is based on the i8259 Programmable Interval Timer (PIT). Another option would have been to use the CMOS 'Real-Time Clock', but it is already in use for the MINIX 3 profiling code [52] and having two sources of timer interrupts would have complicated the kernel's code.

Working of RRES Accounting Although the PIT ticks come at a lower frequency than the TSC cycle counter, the RRES framework can do its work as follows. During initialization RRES calibrates the TSC cycle counter using the CALIBRATE_TSC in order to determine the number of cycles per microsecond. Budget exhaustion and budget replenishment events are expressed in CPU cycles rather than PIT ticks in order to prevent rounding errors in the calculation. This number is reported to the kernel on START_SCHEDULE and START_RECHARGE, respectively, which stores the count in a global variable and compares it to the current cycle counter value on each PIT tick. If the current cycle counter value exceeds the exhaustion or recharging time, the ker-

nel deschedules the task (in the former case only) and sends an RRES_EVENT notification to the user-space RRES manager.

One important decision was at which frequency the TSC counter should be read, that is, the PIT interrupt frequency—since a higher frequency leads to a lower worst-case accounting error. The maximum usable frequency is limited, however, since each PIT interrupt requires reprogramming the timer. After some experimentation we decided to use a PIT frequency of 4000 Hz, which limits RRES accounting error to at most 250 μs. Moreover, task overruns are taken into account by the RRES manager by reading the TSC cycle counter after the RRES_EVENT notification, comparing it with the original deadline, and reducing the task's CPU budget in its next execution frame.

Although RRES accounting works at 4000 Hz, we used a frequency of 500 Hz for the system's normal tick facility. This distinction takes place in the clock task's interrupt handler, which scales the hardware PIT frequency into lower-frequency system-wide ticks, that is, only 1 in every 8 interrupts is transformed into a system tick.

Eliminating CPU Monopolization An important benefit of our design is that denial of service (DoS) attacks that monopolize the CPU [64] are structurally eliminated. By basing accounting on the actual number of CPU cycles used, independent of the PIT ticks, a task can no longer cause another task to be billed by suspending execution just before a PIT tick occurs. In contrast, whenever a task served by RRES stops execution, the RRES manager is informed and the current TSC cycle counter is read to decrease its remaining budget with the number of CPU cycles consumed. Processes that use MINIX 3's standard scheduling facilities are still vulnerable, but real-time tasks and, in fact, any application with stringent timing requirements can use the new RRES framework for temporal protection.

4.4.5 RRES case study

To better clarify how the framework works, we now discuss an example that shows the interactions of the RRES framework, configured to use CBS with hard-reservation mode (CBS-HR). We analyze the sequence of events for two real-

time tasks, T_1 and T_2, producing the schedule shown in Fig. 4.6. Initially, the administrator starts the tasks using the *rres_create* utility. The command entered is

$ rres_create <budget> <period> <binary>

where the request parameters are

<*budget*>: CPU budget given in each period (Q) in μs;
<*period*>: the VRES granularity (P) in μs;
<*binary*>: the application to be managed by RRES.

This request has to be made for both task T_1 and T_2 with parameter $Q_1 = 3000\ \mu s$, $P_1 = 9000\ \mu s$ and $Q_2 = 2000\ \mu s$, $P_2 = 3000\ \mu s$. The sum of the fractions $\frac{Q}{P}$ gives the CPU utilization and is 100% in this example.

For both tasks, the *rres_create* utility forks a new process, sends a CREATE message to the RRES manager to inform it about the new real-time task's parameters, and executes the binary. RRES first checks if the user is authorized and then performs an admission test. Since the CPU utilization does not exceed 100%, RRES accepts the requests, creates two virtual resources R_1 and R_2 with the required parameters, and sends a START_RT_TASK message to the kernel to tell that T_1 and T_2 are real-time tasks from now on. The virtual resources, R_1 and R_2, will be enqueued in RRES' ACTIVE queue, with task T_2 at the head of the queue, since T_2's initial deadline is earlier than that of T_1.

We will now analyze the interactions between the RRES manager and kernel during the execution of tasks T_1 and T_2, which produces the schedule shown in Fig. 4.6. As discussed in Sec. 4.4.4.4, the RRES manager uses the TSC cycle counter for accounting. For reasons of simplicity, however, all times below are expressed in milliseconds.

At time $T = 0$, RRES issues a RRES_SCHEDULE request to the kernel specifying the task to be scheduled, in this case T_2, and the amount of CPU budget, that is, how long the task is allowed to execute, in this case 2. The kernel accepts the RRES request, sets up the time at which the budget is exhausted, and schedules the task in the queue with priority level RT_PRIO.

At time $T = 2$, the kernel notifies RRES about the budget exhaustion of

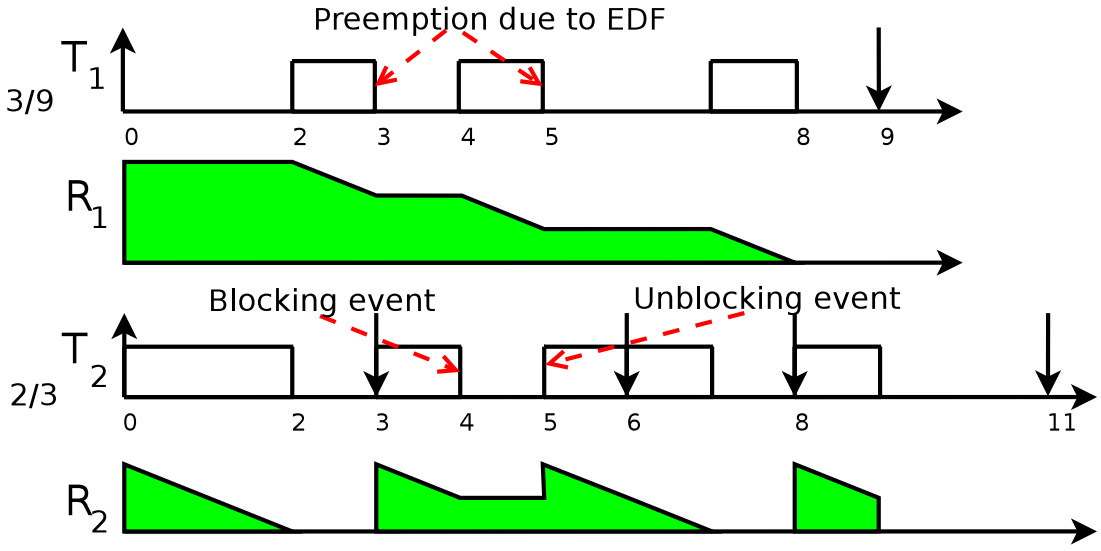

Figure 4.6: Schedule of the case study in milliseconds.

T_2. RRES moves R_2 from the ACTIVE to the RECHARGING queue and, since hard-reservation mode is used, asks the kernel to recharge R_2's budget until the absolute time of R_2's deadline, $T = 3$. RRES also tells the kernel to schedule task T_1 with budget $Q = 3$

At time $T = 3$, the kernel notifies RRES about R_2's budget being recharged, so that RRES moves it from the RECHARGING queue back into the ACTIVE one. Since R_2 has the earliest deadline, T_1 is preempted and RRES asks the kernel to schedule task T_2 with a budget of 2.

At time $T = 4$, task T_2 experiences a blocking event. The kernel notifies RRES, which in turn moves T_2's virtual resource, R_2, to the BLOCKED queue. Then RRES asks the kernel to resume execution of T_1 with a budget of 2.

At time $T = 5$, task T_2 unblocks. RRES is notified by the kernel and computes the test in CBS rule 3. Since the remaining budget $c = 1 \geq (6 - 5)\frac{2}{3} = \frac{2}{3}$ a new deadline is placed at $T = 8$ and the budget is recharged. R_2 is moved to the ACTIVE queue and task T_1 is preempted by T_2.

At time $T = 7$, R_2's budget is exhausted again. RRES is notified by the kernel, moves R_2 to the RECHARGING queue, and tells the kernel to recharge until $T = 8$. RRES also requests the kernel to resume execution of task T_1 with

R_1's remaining budget of 1.

At time $T = 8$, two things happen: R_1's budget is exhausted and R_2's budget is recharged. R_1 is moved to the RECHARGING queue and the kernel is told to recharge the task until R_1's absolute deadline, $T = 9$. In addition, RRES ask the kernel to schedule task T_2 with a budget of 2.

This example shows how a user-space scheduler can do all the work using a small number of interactions with the kernel, obtaining the schedule produced in Fig. 4.6. In the following section we will see how these interactions impose a very limited timing overhead on the system.

4.4.6 Experimental evaluation

In addition to the above case study, we ran several experiments on a prototype implementation to evaluate the RRES framework. The results are presented below.

4.4.6.1 Timing Measurements

As explained in Sec. 4.4.4.4, time accounting is done using the *TSC* cycle counter. The TSC facility is available in both kernel space and user space, allowing RRES to be kept synchronized with the kernel time line. In addition, this enabled precise timing measurements, depending on CPU speed only. The tests were conducted on a Fujitsu-Siemens desktop PC with a 2.8 GHz AMD Athlon64 CPU and 1 GB RAM. None of the tests required to access the disk.

First, we measured the latency introduced by MINIX 3's message passing subsystem, which is independent from the RRES framework. In particular, we measured the time between issuing a request in a user process (just before IPC_SEND) and the moment that the kernel starts working on it (just after IPC_RECEIVE), that is, the time purely spent on delivering the message from the user process to the SYSTEM task. We found a message delivery time of 1.5 μs.

Second, we measured the latency introduced by the RRES framework. These tests were done in the context of the case study discussed in Sec. 4.4.5. We ran several tests and computed the mean result rounded to microsecond precision.

- Time between receiving a *rres_create* command in the RRES framework and the moment that the kernel schedules the new real-time task: 192μs.

- Time between budget exhaustion in the kernel, causing an RRES_EVENT notification processed by the user-space RRES framework, and the moment that the kernel puts the VRES in the recharging state: $43\mu s$.
- Time between detecting a budget-recharged event in the kernel, notifying RRES, and the moment that the kernel reschedules the corresponding task: $41\mu s$.

These results clearly show a very limited overhead imposed by the RRES framework on the system in order to enforce the CBS, CBS-HR and IRIS algorithms.

It is important to realize that these values are not dependent on the presence of other real-time tasks, because (1) the kernel's interrupt handler always preempts running tasks and (2) messages that are exchanged upon RRES events are delivered and handled at the highest priority, as shown in Fig. 4.5.

The measured values have to be compared with the resolution the system is able to grant to the framework. Since time accounting is done at 4000 Hz, the minimum amount of budget and period can, in principle, be $250\mu s$. However, to prevent compromising the requested parameters, they should be at least an order of magnitude larger. Therefore, the budget and period should be set starting from 5–10 ms in practice.

4.4.6.2 Impact on Kernel and User-Space Code

With help of the Source Code Line Counter [19] tool available on the Internet we collected data on the total engineering effort required. The number of executable lines of code for both the standard and modified version of the MINIX 3 kernel are shown in Fig. 4.7. Similar statistics for the new user-space RRES manager are shown in Fig. 4.8

4.4.6.3 RRES Tracer and Simulations

We also created a tool written in Ruby to trace the execution of RRES real-time tasks. The tool parses a log file generated by the RRES server and produces a graphical representation of the scheduling decisions taken.

Fig. 4.10 represents a piece of the scheduling of the task set in Fig. 4.9 that is scheduled according to CBS-HR (CBS with hard reservations); IRIS' time warping is not used. The tasks used are an infinite CPU-bound program (*cpuload*) that performs calculations in a loop and a finite I/O-bound program (*interactive*) that does some work, sleeps one second, and continues calculating. The tracer output shows three aspects:

- *cpuload* continuously triggers CBS' deadline postponement rule, as is clear in the first two task lines where arcs connect consecutive deadlines;
- since *interactive* has a large budget, it can execute whenever there is a free slot, unless it blocks on the sleep() system call;
- at that point, the hard-reservation mode becomes evident, since the two *cpuload* utilities run without time warping (the scheduling is not work-conserving).

Numerous other simulations have been run to verify the behaviour of our implementation in few real cases, but we refrained from including them here due to space limitations.

File	**Standard**	**RRES MINIX 3**	**Delta**
proc.h	99	103	+4
proc.c	482	500	+18
clock.c	115	137	+22
system.c	314	327	+13
rres.h	-	24	+24
rres.c	-	197	+197
do_resres.c	-	131	+131
Total Changes			**+339**

Figure 4.7: Lines of executable code (LoC) for the standard MINIX 3 kernel and the modified version with the RRES framework.

Header Files	LoC
glo.h	42
inc.h	29
proto.h	51
rres.h	106
Header Total	**156**

Source Files	LoC
main.c	158
rres.c	543
rres_kernel.c	254
rres_userspace.c	251
Source Total	**1206**

Figure 4.8: Lines of executable code (LoC) for the RRES server.

Task	Type	Budget (ms)	Period (ms)
cpuload	CPU-bound	100	400
cpuload	CPU-bound	200	2000
interactive	I/O-bound	10000	20000

Figure 4.9: Task set and reservation parameters used for tracer simulation. The execution is shown in Fig. 4.10.

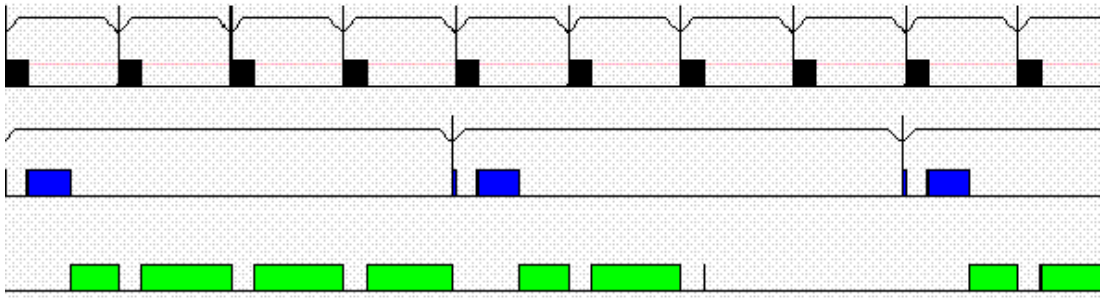

Figure 4.10: Actual schedule executed for the task set of Fig. 4.9 produced by the RRES tracer based on RRES server logs.

4.4.7 Conclusions and future work

MINIX 3 is a dependable multiserver operating system for uniprocessor systems. Its modular design makes it a likely candidate for embedded systems, but MINIX 3 currently lacks real-time support. Therefore, we have enhanced MINIX 3 with *temporal protection* via *resource reservations*. To the best of our knowledge, this had not been done before. Long latencies and slow response times caused by the message passing mechanism were a potential bottleneck, but measurements on a prototype implementation have shown that this effect is very limited and can be

mitigated by carefully designing the framework interactions.

Our resource reservation framework (RRES) implements the CBS, CBS-HR and IRIS resource reservation algorithms and provides temporal protection in order to prevent ordinary users from monopolizing the CPU. Our design enables running soft real-time applications on MINIX 3. The current status is that correct time accounting happens in presence of nonblocking tasks. If blocking events occur, the framework operates correctly under the assumption of short server and driver execution times. Since kernel's generic interrupt handler has a short strictly bounded execution time, limited hard real-time support is provided for tasks that do not rely on the standard MINIX 3 services. In addition, the RRES framework eliminates denial of service (DoS) attacks [64] targeting the scheduler, because time accounting uses the TSC cycle counter independent from the system tick facility.

Work in the context of FRESCOR [7] is in progress to implement a microkernel equivalent of *bandwidth inheritance* [41] algorithm so that the drivers and servers working on behalf of a real-time task can use its RRES parameters during the servicing time. This gives two important benefits, namely, correct time accounting and a very simple resource-access protocol, *priority inheritance*, in order to prevent priority-inversion phenomena. In addition, we intend to analyze the possibility of reserving other resources types, such as file system and network access, through the RRES framework. Success in this area would result in a completely compartmentalized and fully protected resource environment, enabling full hard real-time support.

We will see in the next chapter how we improved this work through the introduction in the context of MINIX 3 of a general resource reservation framework which would allow one to conceive and implement several algorithms very easily.

4.5 Conclusions

In this chapter we have seen how to apply the fundamental real-time concepts seen in the previous chapters to some existing operating systems, as Linux and MINIX 3. We have carefully described some design choices that drove our implementations inside those systems.

Many aspects had to be threaten differently in order to reflect the OS ar-

chitecture we had to work with. In particular, in the context of MINIX 3, the only IPC facility available in the system - the message passing mechanism - mandates several messages exchanges in order to handle every algorithm aspect. In contrast, the monolithic nature of Linux allows a far higher efficiency when fast reactions are needed in order to face every possible situation determined by the real-time scheduler. This comes at the expense of system dependability: being everything in kernel context, every operation might potentially lead to a whole system crash. This is not compatible with a mission-critical real-time system.

Chapter 5

The Generic Resource Reservation Framework

In this chapter we are going to present an entirely new way to look at the Resource Reservation framework which will allow one to conceive, design and implement several different algorithms in a very easy and modular way.

In this sense we are speaking more of a taxonomy than a real programming framework, meaning that we are giving a generic way to describe different algorithms in the same domain.

5.1 State Diagram

We decided to base the framework upon the minimum number of server states we deem capable to describe every possible algorithm, by computing the necessary mathematical and logical operations.

By making use of the properties of this framework, the system designer is able to describe every possible running condition, regardless of the way the actual algorithms enforce it.

As an example, consider the way in which different algorithms put their reclaiming properties in action: some of them let CPU idle time be assigned to currently running servers, some others borrow unused budget from currently inactive servers. We would aim at using just our diagram with its states, events and transitions to describe this particular phase, confining the conditions to start

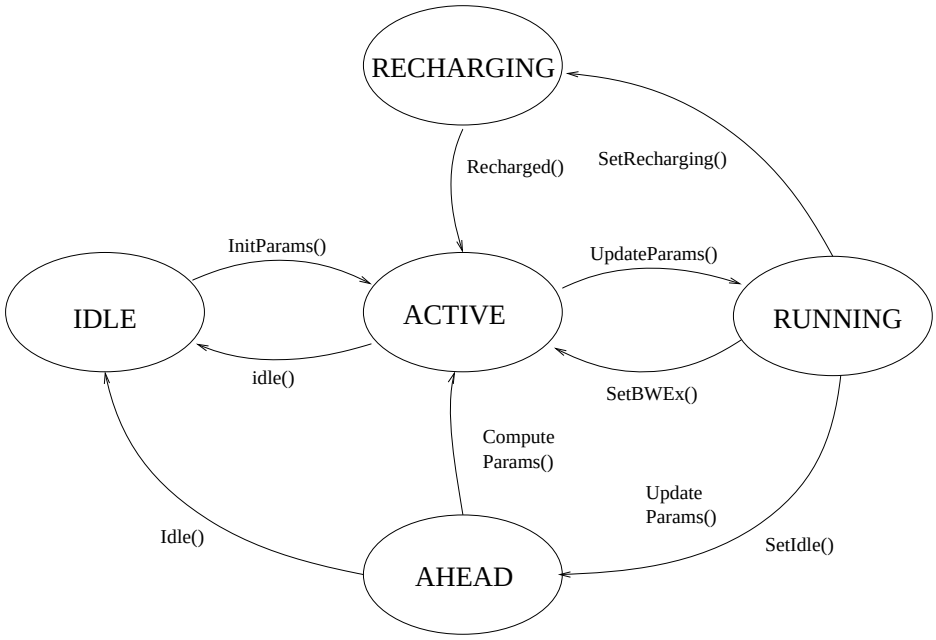

Figure 5.1: Generic framework state diagram

reclaiming at a lower level, along with the specific implementation which enforces this reclaiming.

The state diagram of Figure 5.1 expresses all the possible states of a server. Besides explaining each of them, we will analyze the set of events and consequent transitions towards the considered state.

- *IDLE*, to describe every existing server currently not backlogged, that is which has no ready process to serve (every server just created goes into this state);
- *ACTIVE*, to describe the condition of a server not at the highest priority in the system, with a ready-to-run process in it;
- *RUNNING*, to describe a server currently serving its task, thus decreasing its budget, unless an end condition occurs;
- *RECHARGING*, to describe a backlogged server with no budget, waiting for a budget recharging event to occur in order to start serving its task over;

- *AHEAD*, which is used in case a server is not currently backlogged but has consumed more budget than its "fluid" equivalent (the sources of this budget are of very different nature, as we will see). It is useful to describe several atypical conditions, like budget reclaiming, stealing and other temporary non-standard activities.

The AHEAD state is tightly bound with time $\bar{t}$ of Eq. 3.3: after this time it is definitely possible to consider the current budget parameters as expired and it is safe to switch the current server state towards the IDLE state for future use. Before that time, the server residual budget may be useful for the reclaiming features of several algorithms.

To further refining our framework description, it is necessary to speak of *events* and *transitions*: events determine an action which ends up triggering a transition between an old state and a new one.

A list of possible events follows:

`onTaskBirth` at the end of the creation phase of a new task in the system;

`onTaskReady` when the new task is ready to run (it has backlogged jobs) and the corresponding server has been created;

`onBudgetExhausted` when the current task instance (or job) consummes all the budget reserved for the current server period;

`onBudgetRecharged` in case servers do not get an immediate recharge of their budgets, this happens when the budget is completely recharged;

`onTaskBlock` when the current job experiences a block due to shared resources or explicit signals;

`onTaskUnblock` when, after having been blocked, a job restarts for the blocking condition does not hold any more;

`onTaskDeath` when an application completes its execution or for an abnormal terminating condition (a signal or an exception).

Depending on the events and on the current server state, one of the following transitions may take place:

idle2active occurs on `onTaskBirth` events;

active2running occurs on `onTaskReady` events;

running2active occurs when a higher priority task preempts a lower priority one;

running2ahead occurs on `onTaskBlock` events;

running2recharging occurs on `onBudgetExhausted` events;

gen2idle depending on the current server state, iti occurs when a server is not backlogged (it has no ready jobs);

recharging2active occurs on `onBudgetRecharged` events;

ahead2idle occurs when the current server parameters expire, meaning that it is nonsense to save them for a later use (we will see what it means later on);

ahead2active occurs when a new task instance arrives while the server is in the ahead state and the current parameters may be somewhat maintained or updated;

Finally, there are globally shared operations which must be taken into account in every algorithm, along with specific steps not considered here:

InitParams() used to assign the correct values to the server parameters during the creation phase;

UpdateParams() used to compute the new server parameters following important algorithms events;

ComputeParams() as above, but with the additional computation of a test in order to decide whether UpdateParams() has to be invoked or current parameters may be exploited;

Idle () used when the server is not backlogged any more, that is no more jobs are ready to start;

SetIdle () is used when the current job instance stops, following a special blocking condition like a busy shared resource, an explicit blocking signal or a voluntary sleep;

SetBWExt() used when the current job gets preempted by an higher priority job to save the current parameters after an execution time frame;

SetRecharging() used in case a specific event must be waited for (whether this event is actually of recharging or more general type is left to the specific algorithm implementation);

Recharged(), invoked when the time specified through the SetRecharging() interface has been reached.

We will see, from time to time, how these operations are carried out in the context of the specific algorithms implementations.

5.1.1 Mappings in GRRF

In this section we are going to analyze the way in which the algorithms described in 3.2, get mapped on the state diagram just analyzed.

5.1.1.1 GRRF: CBS

Here the mapping is quite simple:

- the IDLE state maps directly on the CBS IDLE state;
- the ACTIVE state maps directly on the CBS ACTIVE state;
- the RECHARGING state is not used;
- the RUNNING state maps directly on the CBS RUNNING state;
- the AHEAD state is used when a task instance ends and its virtual resource has still some budget. In particular, let q be the current residual budget and U the server utilization, if $(\frac{q}{U} \leq d - t)$, then the server is put and stays in the AHEAD state until $(\frac{q}{U} = d - t)$, time at which the server goes into the IDLE one. As long as it stays in the AHEAD state, if a new task instance is ready to run, it can directly go to the ACTIVE state.

In Figure 5.2 the state diagram equivalent for the CBS case is depicted. Being the concept of budget recharging of no utility, the RECHARGING state has

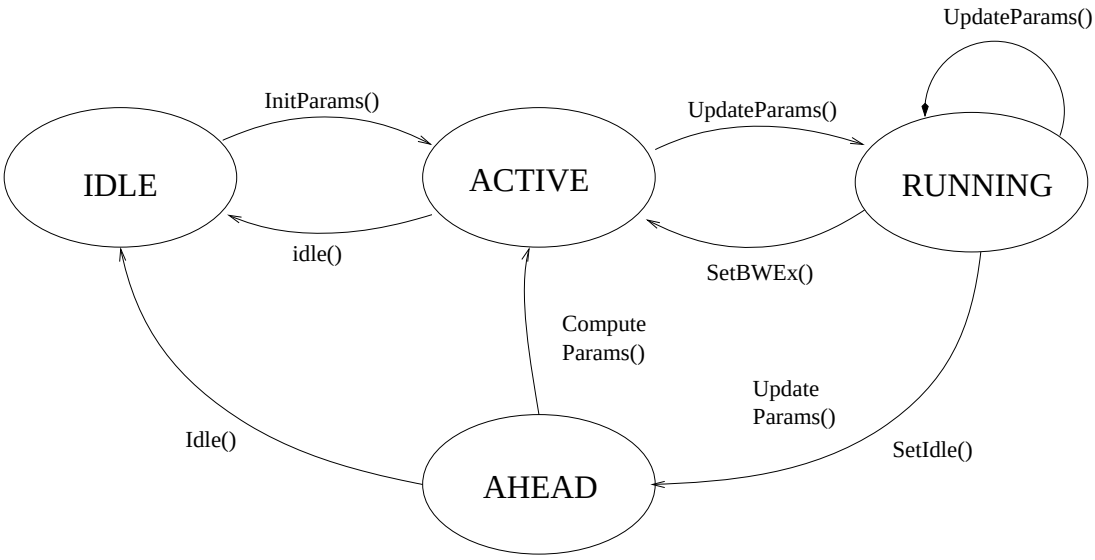

Figure 5.2: The state diagram for the CBS algorithm

been removed and a circular arrow starts from and ends on the RUNNING state, through a simple `updateparameters()` operation.

Let us consider the example of Figure 5.3 and explicitely analyze the distinct phases the framework passes through.

At time $t = 0$ three new task are born, so that three onTaskBirth events are launched. The framework behaves invoking the corresponding generic part of this event handler which, among other activities, takes care of setting up the initial parameters (InitParameters()). It also invokes the corresponding idle2active () which, in this case, is translated into the cbsactive () function call.

At time $t = 2$ τ_A blocks (or, equivalently, its current instance completes). Thus, the framework invokes the onTaskBlock() handler which takes care of moving it to the AHEAD state (through the running2ahead()) and selecting the new server to be put in execution.

An analogous reasoning may be carried on at time $t = 3$, when τ_A wakes up (a new task instance is ready to run). The framework ends up calling the ahead2active() followed by a active2running() which makes τ_A's server start.

At time $t = 9$ a BUDGET_EXHAUSTED event occurs (the task is possibly misbehaving and asking for further execution) , so that the onBudgetExhausted

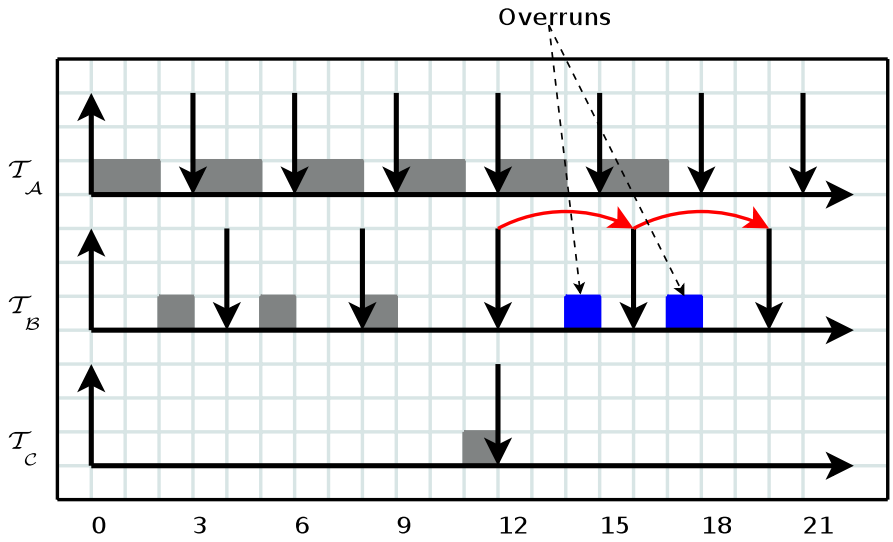

Figure 5.3: A sample CBS schedule to show the GRRF in action

handler gets called. It is immediately translated into the corresponding cbs equivalent, cbsBudgetExhausted(), which works recharging immediately the server budget and postponing the current deadline (at time $t = 16$), according to the algorithm rules. As a matter of fact, this implies a preemption decrease and prevents the task from delaying other tasks execution.

5.1.1.2 GRRF: IRIS

In the IRIS mapping the IDLE, ACTIVE, RUNNING and AHEAD states have exactly the same meaning as in the CBS. The RECHARGING state is directly mapped on the IRIS RECHARGING state.

An important feature of IRIS is the *Time Warping* rule taking place every time there are servers in the RECHARGING and IDLE state only (see Section 3.2.3).

To model this behaviour, it is sufficient to issue a check every time a RUNNING → RECHARGING state transition occurs. If the state-changing server is the last in the RUNNING state and no other one is in the ACTIVE queue, then this rule is triggered and every parameter is updated accordingly.

5.1.1.3 GRRF: GRUB

In GRUB the situation is quite more complex. Besides the standard CBS mappings, we have to describe the algorithm reclaiming capabilities. There are two more issues to be addressed, in this case:

- it is necessary to keep a global current active utilization variable;
- the kernel should notify us whenever it is time to decrease this variable;

The utilization variable should be updated on every important algorithm event, as new servers' creation and task termination, thus we planned to use the kernel notification mechanism employed for all the other events also in this case. Reinterpreting the RECHARGING queue as a WAITING one, it is possible to insert a server in the AHEAD state inside this queue: whenever the kernel will issue a BUDGET_RECHARGED notification message to the RRES manager, by examining the server state, we may conclude it is actually a special event notification and decrese the current active utilization.

In every other situation, GRUB behaves exactly as the standard CBS policy.

5.1.1.4 GRRF: CASH

CASH is likely the most difficult algorithm to implement exploiting our diagram. As GRUB does, it makes use of reclaiming properties to fully exploit the processing power. In order to represent these properties, we are forced to utilize the WAITING queue with fake servers to represent particular time instants to react to.

In CASH, we must keep track of the concept of a global shared queue of remaining, non-exploited budgets, available for ACTIVE servers execution. Furthermore, in every instant t in which this queue is non-empty, the first budget must be decreased at a rate $dc = -dt$, either in case there is an ACTIVE server or not.

So, whenever a RUNNING $\longrightarrow$ AHEAD state transition occurs, we enqueue the server in the AHEAD list, according to its deadline as usual. At this point, if the server is the first in this list, the global budget parameter is set to the current remaining budget of the newly enqueued server and we keep track of the time this setting occurred.

When a AHEAD $\longrightarrow$ ACTIVE transition occurs, we have to compute the CBS test: if the server changing its state is the first one in the AHEAD queue, then there is no need to CBS test it, since its budget got continuosly decreased, so keeping the condition of re-usability valid.

5.2 IMPLEMENTATION

We implemented our framework in the context of MINIX 3 simply extending the work done in [47].

5.2.1 MINIX 3

In Section 4.4 we showed how MINIX 3 can be modified to cope with resource reservations, thus gaining temporal protection properties, the effects of this being an improvement of the overall system dependability. Being all the communications based on message passing mechanism, classical synchronization mechanisms need not to be used, since the synchronous nature of message exchange grants all accesses to shared resources (processes) happen in mutual exclusions.

The modifications introduced at user and kernel level resulted in a platform able to provide the application programmer with a soft real-time, dependable environment. The nature of that implementation did not allow one to easily add new algorithms, practically limiting the application range to a particular type of real-time applications.

By introducing our new framework implementation, though, we grant the system user the possibility to conceive and easily implement new algorithms, more suitable to the kind of applications that will run on the system.

MINIX 3 organization is layer based. Since the kernel scheduling mechanisms of our previous implementation are independent from the particular RRES algorithm chosen, this allowed us to introduce our generic framework at server level, basically modifying our previous RRES server implementation only. This leads to few immediate advantages:

- new features' implementation is isolated at server level;

- policy versus mechanism reciprocal independence allows one to add new scheduling algorithms, without compromising previous implementations;
- every new algorithm must simply implement a limited number of hooks to be fully deployed.

From an implementation point of view, we enriched the **struct** rres_server with a bunch of function pointers:

```
int (*inactive2active)(struct rres_server*);
int (*active2running)(struct rres_server*);
int (*running2active)(struct rres_server*);
int (*running2ahead)(struct rres_server*);
int (*running2recharging)(struct rres_server*);
int (*gen2inactive)(struct rres_server*);
int (*recharging2active)(struct rres_server*);
int (*ahead2inactive)(struct rres_server*);
int (*ahead2active)(struct rres_server*);
int (*admission_test)(struct rres_server*, int);
```

Each of these functions is called whenever a state change event occurs and represents a hook function every algorithm must implement in order to take the actions corresponding to a particular event. As an example, let us analyze the way the CBS-HR algorithm initializes these hooks.

```
        s->algo_type = CBSHR;
        /* ... */
        s->ahead2active = cbshr_start_job;
        /* ... */
```

As it is immediately clear, we are filling the structure fields with functions related to the Hard Reservation mode of the CBS. Finally, let us give a look at one of these hooks implementations.

```
PRIVATE int cbshr_start_job(struct rres_server *s)
{
        /* ... */
        if (cbs_test(s) == RRES_NB)
        {
                /* New parameters must be generated */
                s->tsc_C = mul64u(usec_value, s->Q);
                s->tsc_D = add64(rres_curr_time,
                        mul64u(usec_value, s->P));
        }
        /* The old ones may be used */
        /* ... */
}
```

We are reactivating a job after it reached a stop condition. In this case we have to compute the result of the CBS test in cbs_test() and, depending on its result, enqueue it in the ACTIVE queue with different parameters. Similar code paths may be identified in all the other important algorithm conditions.

From the RRES server point of view, there is the code for events management:

```
case RRES_BUDGET_EXHAUSTED:
        result = onBudgetExhausted(rres_new);
        break;
case RRES_BUDGET_RECHARGED:
        result = onBudgetRecharged(rres_new);
        break;
case RRES_JOB_START:
        result = onJobStart(rres_new);
        break;
case RRES_JOB_END:
        result = onJobEnd(rres_new);
        break;
```

Here, whenever the kernel sends an event message to the RRES server, RRES parses the message and invokes the corresponding function. Job start and end functions represent the condition of unblocking and blocking of a process, respectively (since in a real operating system we have only a few examples of tasks with a real periodic nature).

Lastly, here is a code example for managing one of the previous events:

```
PUBLIC int onJobStart(struct rres_server *s)
{
        /* ... */
        return s->ahead2active(s);
}
```

This is the place where we invoke the specialized version of the transition function. According to the server nature, the correct function implementation gets invoked.

As another example, let us analyze a much more complex algorithm implementat ion as in the case of CASH (see Section 3.2.4). This algorithm enforces a clever reclaiming mechanism by which every active server may use an additional source of budget for its needs.

The following code snippet declares a few global variables, used to implement the reclaiming property.

```
        u64_t gb = 0; /* To hold the current global budget */
```

```
        u64_t gt = 0; /* To hold the time at which the global budget was set */
        u64_t rb = 0; /* To hold the actual residual budget of the borrowing
                                server */
        struct
        rres_server* gs; /* To hold the identity of the server
                                lending its budget */
```

The important time instant to consider for modifications with respect to the original CBS algorithm follow.

Blocks: in this case a new server may become source of extra budget; it gets added to the AHEAD list and, in case it is the first AHEAD server, a fake server is created to represent the lending budget amount.

Unblocks: this is the case when a server wants to restart its service provisioning; in case it was the lending server the residual budget field is updated before the standard CBS test is computed.

In the following code snippet, it is possible to understand how an event is inserted in the WAITING servers' queue.

```
PRIVATE int rres_prepfakesrv(s)
struct rres_server *s; /* The server which the fake one has to conform to */
{
        struct rres_server *rs;
        if(gs != NULL)
                /* Let's set up the former lending residual budget */
                gs->tsc_C = sub64(gb, sub64(rres_curr_time, gt));
        /* Let's update the lending server identity with the new one */
        gs = s;
        /* Let's update the global budget */
        gb = s->tsc_C;
        /* Finally, let's update the global time */
        gt = rres_curr_time;
        /* Check for running server: updating its budget, by means of
         * the rres_check_queue function it will be scheduled for the
        * correct global budget time */
        if((rs = p_srvs[RUNNING]) != NULL)
                s->tsc_C = gb;
        rs = malloc(sizeof(struct rres_server));
        rs->task_endpt = s->task_endpt;
        rs->rres_status = WAITING;
        rs->tsc_D = add64(rres_curr_time, s->tsc_C);
        rres_insert_server(rs, WAITING);

        return RRES_OK;
}
```

When the first of these servers expires (its current deadline gets reached), if its status field is not WAITING but AHEAD, a special handling procedure is invoked to correctly update the environment, as it happens when the following code is invoked.

```
if(s->rres_status == AHEAD)
{
        /* A fake server budget got exhausted .
                * We must:
                * - delete the fake server
                * - update the running server parameters
                * - create the new fake server
        * - put the ahead server in the inactive queue */
        struct rres_server* rs =
                rres_find_server_by_task(s->task_endpt, AHEAD);
        if(rs == NULL)
                return RRES_INVSERVER;
        result = rres_insert_server(rs, INACTIVE);
        if(p_srvs[AHEAD] != NULL)
                rres_prepfakesrv(p_srvs[AHEAD]);
        free(s);

        return RRES_OK;
}
```

The WAITING queue mechanism needs the insertion of a fake **struct** rres_server which is able to simulate an event queue. The kernel acts as the event time signaling mechanism.

Similar mechanisms may be exploited to implement the whole variety of RRES algorithms. When a reclaiming property must be enforced, the WAITING queue and event insertion within it is the way to go.

5.3 Conclusions and future work

In this chapter, we have proposed a completely new approach to the problem of resource reservation algorithms design and implementation, by exploiting a new taxonomy made up by a 5-states diagram. The equivalence in expressive power with respect to every other algorithm's state diagram (where this is available) has not been proven by analytical results and formal procedures, but the high number of succesful experimental mappings shown so far can reasonably mean we are on the right way to go.

A particular mention must go to the algorithms reclaiming capabilities: since they are of very different nature, in order to correctly implement them, we made use either of the AHEAD queue or the AHEAD status in the WAITING queue. We may enforce every kind of special behaviour by simply inserting a fake server in this queue to be extracted later on, upon a `BUDGET_RECHARGED` event reception: being this server in a state different from WAITING we may recognize a special condition to be accounted for.

A lot more work must be done to take into account, through our state diagram, other issues like resource sharing access protocols, multiprocessor scheduling and other types of resources to be reserved. Nevertheless, we are firmly convinced this is a promising and very effective way to abstract both the resource and the reserving algorithm.

Chapter 6

Conclusions

During my four years of research I could experiment many failed occasions to flawlessly move from the theoretical field to the applied one. I saw many excellent algorithms fail in the real case due to unexpected problems, really hard to be taken into account at design time. In fact, this is a very common mistake made in the research domain: when conceiving a new algorithm to start from ideal conditions with negligible latencies, with all planned and expected events occurring in between. This unavoidably leads to uncertainty when moving to a real platform with many unforeseen events or, even worse, hardware failures.

It is from this thought that the idea of a work on the boundary between the two domains was born. In this study we started from the real and concrete world of operating systems and the mostly theoretical of real-time algorithms to gradually move on a ground in between. In particular, we considered and analyzed many existing and widely adopted platforms, and studied every possible way and consequence of possibly introducing real-time mechanisms within them.

We then chose to focus mainly on two of the foremost candidates in the OS scene, namely Linux and Minix 3 for several reasons:

- they are open-source and, as such, it is possible to provide them with additional functionalities;
- they are general purpose, so their application domain is potentially unlimited;
- they represent two very different architectural choices and this constituted

a very important chance to demonstrate the flexibility of Resource Reservation in general and of our framework in particular.

We provided a lot of details about introducing real-time shaped behaviours inside these systems, because we deem this activity one of the most delicate at all. From the implementation point of view, a lot of details must be taken into account: dependability was only one of our primary concerns, whereas reduced latencies and optimal performances are key benefits of such systems.

In the context of Linux, we mainly took into consideration the possibility of actively managing the resource sharing aspect. If not taken into consideration, most assumptions about the time each process runs might be invalidated: to this end, we conceived and implemented an improved resource reservation scheme which allows one to avoid specifying the maximum time length a process will hold a resource.

We finally faced the problem of designing a general taxonomy which would help system developers having to provide a system with Resource Reservation algorithms and capabilities. This framework proved itself a very convenient approach to the complex problem of resource reservation based scheduling systems. Its flexibility was proven by practically implementing it within Minix 3: no assumptions were made about the underlying scheduling system. This allows the system developer to abstract the Resource Reservation algorithms implementation from the specific mechanisms found at kernel level and to focus on the algorithm itself only. As a matter of fact, by implementing few important callbacks, almost every kind of algorithm may implemented at this level.

We are actively discussing about the problem of further developing this taxonomy to include the description of resource sharing aspects and of the most advanced capabilities of modern Resource Reservation algorithms. At the same time, we are developing the very same taxonomy in the context of very different platforms, like Linux and RT-SIM, to have further details to work on in order to refine implementation techniques of what we deem a very useful and important piece of software.

References

[1] ADA-2005 interface language specification. http://www.adaic.org/standards/ada05.html.

[2] Adaptive Domain Environment for Operating Systems. http://home.gna.org/adeos/.

[3] Apple macosx leopard homepage. http://www.apple.com/macosx/.

[4] The arinc homepage. http://www.arinc.com/.

[5] Arinc homepage. https://www.arinc.com.

[6] Cygwin homepage. http://www.cygwin.com/.

[7] FRESCOR - Framework for Real-time Embedded Systems based on COntRacts. http://www.frescor.org.

[8] Ingo Molnar's RT Tree. Available online.

[9] Linux: High-res timers and tickless kernel. http://kerneltrap.org/node/6750.

[10] Linuxworks homepage. http://www.lynuxworks.com/.

[11] List of embedded Operating Systems. http://en.wikipedia.org/wiki/List_of_operatin

[12] Microsoft windows vista homepage. http://www.microsoft.com/windows/windows-vista/

[13] Modular Scheduler Core and Completely Fair Scheduler. http://lkml.org/lkml/2007/4/13/180.

[14] The osek/vdx homepage. http://www.osek-vdx.org/.

[15] Programming Real-Time with Ada 2005. http://www.embedded.com/showArticle.jhtml?articleID=192503587.

[16] Qnx homepage. http://www.qnx.com/.

[17] RTAI home page. https://www.rtai.org/.

[18] RTLinux home page. http://www.rtlinux.org.

[19] Sclc.pl - the Source Code Line Counter. Available online.

[20] The Real-Time Driver Model. http://www.xenomai.org/documentation/trunk/html/api/

[21] The Windriver VXWorks homepage. http://www.windriver.com/products/vxworks/.

[22] Tinyos homepage. http://www.tinyos.net/.

[23] Wikipage for posix. http://en.wikipedia.org/wiki/POSIX.

[24] Wikipage for system. http://en.wikipedia.org/wiki/System.

[25] XENOMAI home page. http://www.xenomai.org.

[26] Luca Abeni. Server mechanisms for multimedia applications. Technical Report RETIS TR98-01, Scuola Superiore S. Anna, 1998.

[27] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proc. IEEE Real-Time Systems Symposium*, Madrid, Spain, 1998.

[28] T. P. Baker. A stack-based allocation policy for realtime processes. In *Proc. IEEE Real Time Systems Symposium*, 1990.

[29] T. P. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems*, (3), 1991.

[30] Sanjoy Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 137–144, Palma de Mallorca, Balearic Islands, Spain, July 2005. IEEE Computer Society Press.

[31] Giorgio Buttazzo and Luca Abeni. Adaptive workload management through elastic scheduling. *Real-Time Syst.*, **23**(1-2):7–24, 2002.

[32] Giorgio C. Buttazzo. Rate monotonic vs. edf: judgment day. *Real-Time Syst.*, **29**(1):5–26, 2005.

[33] Marco Caccamo, Giorgio Buttazzo, and Lui Sha. Capacity sharing for overrun control. In *Proc. 21st IEEE Real-Time Systems Symposium*, pages 295–304, 2000.

[34] Allen B. Downey. *The Little Book of Semaphores*. Green Tea Press, 2007.

[35] Ulrich Drepper and Ingo Molnar. The native posix thread library for linux. 2005.

[36] Cl.-J Hamann, L. Reuther, J. Wolter, and H.Härtig. Quality-Assuring Scheduling. Technical report, TU Dresden, 2006.

[37] Cl.-J. Hamann, Michael Roitzsch, Lars Reuther, Jean Wolter, and Hermann Härtig. Probabilistic admission control to govern real-time systems under overload. In *Proc. 19th Euromicro Conf. on Real-Time Sys.*, 2007.

[38] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Construction of a Highly Dependable Operating System. In *Proc. 6th European Dependable Computing Conf.*, 2006.

[39] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure resilience for Device Drivers. In *Proc. 37th Int'l Conf. on Dependable Systems and Networks*, 2007.

[40] Hiroyuki Kaneko, John A. Stankovic, Subhabrata Sen, and Krithi Ramamritham. Integrated scheduling of multimedia and hard real-time tasks. In *Proc. IEEE Real-Time Systems Symposium*, 1996.

[41] G. Lamastra, G. Lipari, and L. Abeni. A bandwidth inheritance algorithm for real-time task synchronization in open systems. In *Proc. 22nd IEEE Real-Time Systems Symposium*, 2001.

[42] Jochen Liedtke. Toward real microkernels. *CACM*, **39**(9):70–77, 1996.

[43] G. Lipari and S. Baruah. Greedy reclamation of unued bandwidth in constant-bandwidth servers, 2000.

[44] Giuseppe Lipari and Enrico Bini. A methodology for designing hierarchical scheduling systems. *J. Embedded Comput.*, **1**(2):257–269, 2005.

[45] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, **20**(1):46–61, 1973.

[46] Chung Laung Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery*, **20**(1):46–61, January 1973.

[47] Antonio Mancina, Giuseppe Lipari, Jorrit N. Herder, Ben Gras, and Andrew S. Tanenbaum. Enhancing a Dependable Multiserver Operating System with Temporal Protection via Resource Reservations. In *Proc. 16th International Conference on Real-Time and Network Systems (RTNS'08)*, Rennes, France, October 2008.

[48] Luca Marzario, Giuseppe Lipari, Patricia Balbastre, and Alfons Crespo. Iris: A new reclaiming algorithm for server-based real-time systems. In *Proc. IEEE Real-Time and Embedded Techn. and App. Symp.*, 2004.

[49] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: An Abstraction for Managing Processor Usage. In *Proc. 4th Workshop on Workstation Operating Systems*, 1993.

[50] Cliff Mercer, Ragunathan Rajkumar, and Jim Zelenka. Temporal protection in real-time operating systems. In *11th IEEE Workshop on Real-Time Operating Systems and Software*, 1994.

[51] Cliff W. Mercer, Ragunathan Rajkumar, and Jim Zelenka. Temporal Protection in Real-Time Operating Systems. In *Proc. 11th IEEE Workshop on Real-Time Operating Systems and Software*, 1994.

[52] Rogier Meurs. *Building Performance Measurement Tools for the MINIX 3 Operating System*. Master's thesis. Vrije Universiteit, Amsterdam, 2006.

[53] Luigi Palopoli, Tommaso Cucinotta, Luca Marzario, and Giuseppe Lipari. AQuoSA — adaptive quality of service architecture. *Software – Practice and Experience*, **39**(1):1–31, 2009.

[54] Luigi Palopoli, Tommaso Cucinotta, Luca Marzario, Antonio Mancina, and Paolo Valente. "A unified framework for managing different resources with QoS guarantees". In *"Proc. of the first International Workshop on Operating System Platforms for Embedded Real-Time Applications (OSPERT 2005)"*, "Palma de Mallorca, Balearic Islands, Spain", 2005.

[55] Pablo Andrs Pessolani. *MINIX4RT: A Real-Time Operating System Based on MINIX*. Master's thesis. Universidad Nacional de La Plata, 2006.

[56] Ragunathan Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems. In *Proc. Conf. on Multimedia Comp. and Netw.*, 1998.

[57] P. Rogina and G. Wainer. New real-time extensions to the minix operating system. In *Proc. of 5th Int. Conf. on Information Systems Analysis and Synthesis*, 1999.

[58] P. Rogina and G. Wainer. Extending rt-minix with fault tolerance capabilities. In *Proc. Latin-American Conf. on Informatics*, 2001.

[59] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, **39**(9):1175–1185, September 1990.

[60] M. Spuri and G. C. Buttazzo. Efficient aperiodic service under the earliest deadline scheduling. In *Proc. IEEE Real-Time Systems Symposium*, 1994.

[61] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[62] Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao. Real-Time Mach: Towards Predictable Real-Time Systems. In *Proc. USENIX 1990 Mach Workshop*, 1990.

[63] Lisa A. Torrey, Joyce Coleman, and Barton P. Miller. A comparison of interactivity in the linux 2.6 scheduler and an mlfq scheduler. *Softw. Pract. Exper.*, **37**(4):347–364, 2007.

[64] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Secretly Monopolizing the CPU Without Superuser Privileges. In *USENIX Security*, 2007.

[65] Victor Yodaiken. Against priority inheritance. Technical report, Finite State Machine Labs, June 2002.